A Low-Cost Part-Task Flight Training System: An Application of a Head Mounted Display

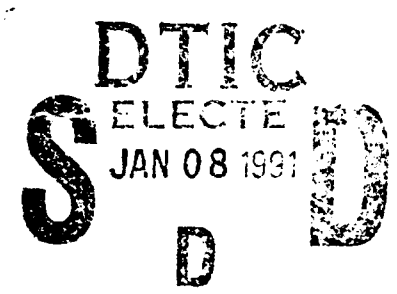
THESIS

David Anthony Dahn
Captain, USAF

AFIT/GCE/ENG/90D-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/90D-01



A Low-Cost Part-Task Flight Training System: An Application of a Head Mounted Display

THESIS

David Anthony Dahn
Captain, USAF

AFIT/GCE/ENG/90D-01

AFIT/GCE/ENG/90D-01

A Low-Cost Part-Task Flight Training System: An Application of a Head Mounted Display

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

David Anthony Dahn, B.S.
Captain, USAF

December, 1990

Approved for public release; distribution unlimited

Preface

Numerous people deserve thanks for their special contributions, aid, and encouragement that made this thesis a success.

First and foremost, a grand thanks to my wife – Carla, and children – Andy and Chelsey, for their love, patience and encouragement.

I thank my sponsors, the AFHRL Operations Training Division and the AAMRL Human Engineering Division. Without their funding and equipment loans, this research could not have been attempted. Special thanks to MSgt Kashmere and Mr Don Jones, the two individuals at Williams AFB that finally got my host computer system shipped!

Many thanks to Major Phil Amburn, my thesis advisor. When I needed driving, he was always able to provide guidance. I also extend special mention of my thesis readers, Maj David Umphress and Maj Marty Stytz, for providing corrections and useful advice for improving the document.

Special thanks to Capt Ed Williams who willingly shared his expertise whenever needed, even if it interrupted his research work. Ed, you are one of the few true technical gurus. Ed helped resolve the hardware problems, installed UNIX on the system, handled all system administrator tasks throughout the development, and wrote the UNIX joystick driver and supporting library. His efforts allowed me to focus on the application layer of the software developed under this thesis.

Finally, I want to thank Mr. Charlie Powers, Capt Dean Campbell, and Ms Pam Young who took care of my urgent equipment and purchase requests. I also extend thanks to AFIT/SC for arranging to fix the equipment that was broken in transit from Williams AFB.

David Anthony Dahn

Table of Contents

List of Figures

List of Tables

Abstract

Computer Image Generators (CIG) driving high performance flight simulators used in training pilots are expensive. This project investigated whether a small class of these simulators that focus on task-specific training could be hosted using much cheaper simulator systems. Effective training systems provide a wide visual field-of-view through the use of tesselated CRTs or dome simulator projectors. These display systems require graphics processing support from expensive CIGs with multiple graphics channels.

A promising technology that could help reduce the costs of these flight simulators are head-mounted display (HMD) systems. Simple virtual world interfaces using HMD technology require only one graphics channel. With a single graphics channel requirement, potential exists to use low cost CIGs.

To investigate the feasibility of using HMDs and CIGs in a low cost part-task trainer, we created a prototype system. Our approach was to build a virtual world interface using a HMD to an existing flight simulator application. This allowed a CIG with only one graphics channel to drive the display. To investigate which class of computing platform was suitable for use as the CIG, a cooperative thesis effort was launched to host the simulator on a mini-computer and micro-computer platform. The appropriate CIG could then be determined through demonstration and comparison between the systems.

We implemented the virtual world interface for the Silicon Graphics' Flight program on an 80386/80387 PC-AT enhanced with a high performance graphics engine and a Silicon Graphics IRIS 4D/85 GT. This project focused on the PC-AT with a Real World Graphics Ltd. PC *Reality* board, containing two Intel i860 RISC processors, as the graphics engine.

A software emulation library was built to transform the Silicon Graphics graph-

ical reference model and function calls to the PC Reality PHIGS PLUS graphical reference model and function calls. This emulation allowed a nearly seamless interface to support porting the application program from the Silicon Graphics machine to the PC.

A virtual world interface using the Air Force Institute of Technology's head-mounted display II was implemented. Joysticks provided a throttle and stick interface decoupling the user from the keyboard. This allowed a user to don the helmet and fly the flight simulator as if he were the pilot of the aircraft simulator.

One focus was to determine whether the PC environment was mature enough to support this approach. The specific question we tried to answer was whether the flight simulator could be programmed on the PC using a classic workstation approach (written in a high order language using a standard three dimensional graphical reference model). The measure of success was whether the simulator could provide a frame update rate of 15 frames per second or better for Z-buffered, flat-shaded polygons.

The results were short of the requirement. Our conclusion is that the price performance ratio in terms of frames per second was better for the higher priced mini-computer approach than the super-charged PC approach.

A Low-Cost Part-Task Flight Training System: An Application of a Head Mounted Display

I. Introduction

This research investigated using a head-mounted display (HMD) system as a viewing device for a low-cost, part-task (or task specific) flight training system. The head-mounted display provided a 360° (full-vision) color view that could be used as part of a low fidelity flight simulator. The user slips into an enclosed display system that places him (or her) into a virtual world cockpit of a high performance fighter aircraft. The 'pilot' flies the aircraft using joysticks connected as a throttle and stick.

One goal of this investigation was to determine whether a low-cost, single-channel computer image generator (CIG) and HMD could provide a suitable full-vision, part-task flight trainer. Another goal was to determine which class of computer (mini or micro) could support the graphical output requirements.

This project focused on the micro-computer approach. The objective was to determine whether a personal computer (PC) with a state-of-art-graphics engine, was powerful enough to use as the single-channel CIG. Another objective was to determine whether a workstation approach[1] to graphical programming could be used on a current state-of-the-art PC platform.

1.1 Key Terms

Part-Task: Flying skills can be broken down into parts called tasks. For example, landing an aircraft, take-off, and formation flying all require a set of loosely related tasks that are required flying skills. These tasks can be

[1]Software written in a high order language using a classic 3D graphics reference model.

further broken down into subcomponents called part-tasks. Part-tasks are usually those tasks that need to be practiced until they become a reflex reaction for the pilot. Good examples of part-task skills are aerial gunnery and a pilot's evasive maneuver response to a radar warning receiver.

Virtual World: The virtual world is a three dimensional computer graphics depiction of the world on a two dimensional screen. The user enters the virtual world by entering a display device that encloses the user, totally replacing everything he normally sees in the real world by computer generated imagery (CGI). Full-dome simulators and head-mounted displays are two examples of virtual world environments. The virtual world experience was expanded in recent years to include other man-machine interfaces to meet the challenge issued by Ivan Sutherland in 1965 when describing what was meant by a 'virtual world'. "The screen is a window through which one sees a virtual world. The challenge is to make that world look real, act real, sound real, feel real"(43). These man-machine interface advancements have included technologies such as a virtual glove that can be used to pick up "artificial" (computer-generated) objects and move them to a new location; surround sound to let a virtual world user hear sound from the direction of generation (for example, if a pilot's wingman off his left wing talks, the pilot hears the sound in his left ear); and tactile feedback where a pilot would feel compression on the body if the aircraft accelerates or feel pressure on the end of a finger if a virtual button is pressed.

Full-Vision: Full-vision refers to a user's ability to see everything around him. For our research, full-vision does not mean a CGI view is provided for a person's full peripheral viewing ability, only that the user perceives he has a sufficiently wide field-of-view to prevent restricted vision. R.E. Lambert identified a 60 degree square field-of-view as an acceptable lower limit when using a HMD for visual air combat(31:6). Full-vision also means a person can turn around and see what is behind him without any special action on his part.

1.2 Background

The Air Force Human Resources Laboratory, Operations Training Division, and the Armstrong Aerospace Medical Research Laboratory, Human Engineering Division sponsored this thesis investigation. Both organizations are actively pursuing the use of a head-mounted display for diverse Air Force applications.

The Air Force Human Resources Laboratory (AFHRL) is responsible for developing flight simulator systems to support flight training. AFHRL and the Naval Air Development Center have presented evidence that shows pilots trained in flight simulators prior to actually entering an aircraft perform significantly better than those that start in the cockpit(1, 8, 14, 28). The AFHRL has developed realistic dome simulators using the Advanced Simulator for Pilot Training (ASPT) software. These simulators provide realistic simulated flight dynamics for the specific aircraft being flown. However, these simulators cost too much for widespread use. The AFHRL is now looking at the possibility of conducting training, most likely part-task training, in lower-cost, lower-fidelity, full-vision simulators. The full-vision capability is considered critical because of the intense visual activity experienced in air-to-air combat(14:A40). The lower-fidelity approach is feasible because, as Hal Geltmacher of the AFHRL writes:

> The concept is based on the fact that only 130,000 visual resolution elements or pixels can be observed by the eye at any instant in time. This concept, coupled with the inability of the human to distinguish intermittent visual occurrences if they occur at moderate rates (30 HZ), leads to the conclusion that one should be able to generate a wide-field high-resolution display with no more information processing requirement than those of a conventional 525-line television system.(14:A42)

The Air Force Institute of Technology's third generation HMD, now under development, may be able to provide sufficient resolution to support validating the premises of this concept.

Armstrong Aerospace Medical Laboratory (AAMRL) is responsible for advanced technology human factors developments supporting the improvement of the pilot/cockpit interface. AAMRL has an on-going research effort to develop a "super cockpit." The basic idea behind the super cockpit is to provide the pilot with virtual world displays on his helmet that would allow him to see a graphical depiction of what he would normally see in flight. Representations of the location of Surface-to-Air Missile sights and their threat regions, even if they are beyond visual range, could also be depicted. Another advantage AAMRL has identified for a virtual cockpit is to let a pilot switch from one aircraft type to another with little or no training. Aircraft today use electro-mechanical instruments and gauges located in positions that are unique to each individual aircraft type. In a virtual cockpit, the pilot could load his cockpit software during pre-flight and conceptually fly an aircraft independent of its type. AAMRL has defined three major components of a "super cockpit": first, a head or helmet-mounted display to provide the full-vision virtual display system; second, a means to provide audio feedback to the pilot so he can determine from which direction the sound was generated; and finally, a means to provide tactile feedback to the pilot when he is pressing a button on a virtual world panel. The principal component is the HMD. AAMRL is also interested in other Air Force applications of HMDs that may significantly benefit or enhance a users' ability to view or manipulate graphical data.

Two such applications are Command and Control and Mission Planning. The application of a HMD in Command and Control would allow a planner to enter a virtual world showing the battle field and all known defenses. This planner could then feasibly move aircraft packages from one ingress point to another, or a jamming platform from one point to another, and then study the effects to determine if there are better force postures that could significantly improve the war-fighting ability. The HMD could also support Mission Planning by allowing the pilot to fly a simulation of his mission before ever entering the aircraft. The pilot could then identify any

additional items missed in the preliminary plan. Networking several low-cost HMD simulators together would allow a strike team to fly their mission together. This mission practice concept is expected to improve the pilot's survival rate during the actual mission(1, 4, 17).

The Air Force Institute of Technology has been developing a virtual world laboratory to support on-going research to extend the application of head-mounted displays and other virtual world interfaces. To date, the Institute has developed two generations of head-mounted displays(11, 38) and a supporting virtual world environment(11). These displays were used to provide a mission replay capability of RED FLAG exercise data(38) and a command and control application allowing a pilot to pre-fly a mission defined in an Air Tasking Order(47).

1.3 Thesis Statement

A PC-AT with a graphics coprocessor and the Institute's HMD provide a low-cost alternative to existing full-dome, task-specific training systems.

Existing full-vision simulators are expensive. The use of HMDs is a less expensive alternative to full-dome simulator systems. Alternative HMD designs are now being investigated by numerous research organizations. These research efforts have focused on the feasibility of using the HMDs as an alternative to full-dome simulators without focusing on the cost of the computer graphics image generator used to create the virtual world images. We emphasize this focus through demonstration.

1.4 Scope

This research emphasized the design and development of a low-cost simulation station based on a PC compatible computer and head mounted display. The research investigated the creation of a simulation capability with a full 360° field-of-view through the use of head tracking and head-mounted displays. Commercially-available computer graphics hardware provided the computer generated imagery.

To support this research the Institute's HMD was connected to an Intel 80386 based PC-AT, with a supplementary graphics engine, to provide a virtual world flight simulator.

The flight simulator software was an adaptation of the Dog[2] program. Dog is an interactive flight simulator controlled by a keyboard and mouse with cockpit displays and out-the-window scenery displayed on a monitor. The software is written in C using the Silicon Graphics 3D graphical reference model.

The measure of success was to implement a virtual flight simulator that used a HMD and throttle and stick interface that worked as well as the existing 2D display and keyboard interface. However, instead of pressing a key to get a different view, like off the left wing, the user just looks left. Rather than pressing 't' (for throttle) to increase power, a throttle is used; and rather than using a computer mouse to steer the aircraft, a joystick is used.

Many other related simulator issues were raised by this research but were not included in this investigation. Some of these issues are:

- Could even lower cost platforms be used? Could lower speed or less powerful graphics coprocessors be used to implement a comparable simulator? Could a PC with a standard VGA card be used rather than a graphics coprocessor?
- Could a higher performance (better flight dynamics and more realistic out-the-window scenery and in-cockpit displays) simulator have been used?
- What trade-offs concerning processing power versus simulator requirements could be made to enhance performance?
- Is the HMD image fidelity sufficient for flight training?
- Could a better man-machine interface be developed?

[2]Dog is a copyright of Silicon Graphics Incorporated.

1.5 Assumptions

The following equipment was assumed:

1. A 20 MHZ 386 PC-AT with sufficient disk storage space, ethernet card, game card, two joy sticks, a mouse, and a supplementary graphics engine capable of rapidly displaying Gouraud shaded polygons in EIA RS-170 NTSC Color System format.
2. The UNIX System V operating system, with software development extensions. A 'C' language compiler, linker, loader was required.
3. Flight simulator software written in 'C' code was a requirement. Starting from scratch was unrealistic given the time available.
4. A Polhemus 3Space Isotrak magnetic tracking device to track user head movement. This assumption includes the supporting software required to operate the device.
5. A cockpit mockup which would hold the equipment.

1.6 General Approach

This thesis investigation was a cooperative effort with Capt Phil Platt. Capt Platt implemented a similar HMD part-task simulator on a more expensive Silicon Graphics IRIS 4D/85 GT computer platform. He demonstrated the implementation of a virtual world simulator on a medium cost graphics workstation, whereas I was investigating whether a much lower cost approach to a HMD part-task simulator could provide comparable performance.

The general approach for this thesis consisted of three major steps:

1. First, the literature search and equipment requirements (both software and hardware) were identified. Through the information provided from the litera-

ture search I established requirements for the part-task simulator and identified the specific niche this application could fill in supporting the Air Force mission.

2. Second, the equipment was integrated and the software tools installed on the host machine. An 80386/80387 PC supplemented by a Real World Graphics Ltd. PC *Reality*[3] graphics board was used. The PC Reality provided two Intel i860s on a single PC board and a software graphics library for the C programming language. The PC Reality board served as the graphics pipeline with the only programmer interface being through calls to the graphics library. The board provided a configurable frame buffer that supplied the required NTSC signals for the head-mounted display.

3. Third, the part-task simulator software was then developed. This task consisted of modifying the Silicon Graphics Dog software to include a different cockpit display and run on a platform other than the Silicon Graphics 4D. A software emulation library was written to translate function calls to the Silicon Graphics hardware graphics pipeline into library calls to the Real World Graphics Ltd. graphics library. This Real World Graphics library was based on the Programmer's Hierarchical Interactive Graphics System Plus Lumière Und Surfaces (PHIGS PLUS or PHIGS+) standard. With the change in graphical reference models, the goal was to have the emulation library handle all the transformations from the Silicon Graphics Reference model to the PHIGS+ reference model.

1.7 Summary

This thesis extended the Institute's research into an interactive virtual world environment using low-cost computing resources. This investigation was a natural evolution of previous research conducted at the Air Force Institute of Technology's Virtual World Graphics Laboratory(11, 27, 32, 38, 47).

[3]Reality is a copyright of Real World Graphics Ltd.

To support the on-going research, a virtual world flight simulator prototype was developed for a PC AT supplemented with a high performance graphics engine.

1.8 Thesis Overview

This thesis is composed of five chapters. The first chapter presented the Air Force Institute of Technology's virtual world research goals and the evolving capabilities of the Institute to accomplish research supporting those goals. The second chapter contains a literature review identifying various needs for flight simulators. Chapter two also contains background information on the state of head-mounted display technology development and other areas of interest directly supporting the development of the part-task simulator. Chapter three summarizes the requirements defined during the developmental requirements analysis. Chapter four provides the specific system design. Finally, chapter five discusses the actual implementation followed by a subjective assessment of the utility and performance attained by the low-cost flight simulator approach. Chapter five also contains conclusions for this project and recommendations for future research.

II. Literature Review

There were three areas of interest which had to be researched before any system design efforts could begin. These areas are 1) investigating the history of Air Force interest in flight simulator development and the missions these flight simulators satisfy, 2) conducting a survey of Head or Helmet Mounted Display systems and 3) researching low-cost alternatives for the architecture of flight simulator equipment.

2.1 Purposes of Flight Simulation

Flight simulators can provide significant contributions to two unrelated, but crucial aspects of flying: Pilot Training and Research & Development of Aircraft Systems. This review presents support for each aspect.

2.1.1 Pilot Training. Pilot training using flight training simulators significantly improves the pilot's performance in the cockpit(1, 17). Both Air Force and Navy studies show significant improvements in specific flying tasks when pilots have practiced that task in a flight simulator prior to actually climbing into the aircraft(14, 28). Flying tasks are normally mastered more quickly and at a higher skill-level if the subcomponents (part-task) of a flying task are broken down and practiced separately or in a different order than normal(17:103).

Four additional benefits are derived from using flight simulators to conduct flight training. The most significant benefit is the additional safety afforded the pilot when practicing dangerous maneuvers. Aircraft emergencies, system failures, and other potentially disastrous events can be practiced with no risk to the pilot or the aircraft. Another benefit is the concentration of flying experience gained in comparison to training in an actual aircraft. An hour of flying time in air combat maneuvers may only allow three or four engagements, with each engagement only lasting a couple of minutes. With a flight simulator, 20 or more engagements can

occur in the same hour. A third benefit is the ability to use flight simulators to evaluate different training programs through repeatable training situations. Training instructors could compare different programs and identify the advantages and disadvantages of each training curriculum. The most effective components could be used to develop a *best* program. A fourth significant benefit is the low cost of operating a flight simulator in comparison to training the pilot in an actual aircraft in flight(17, 49).

The benefits already outlined are noteworthy in their own merit, but the most significant impact of the flight simulator may very well be the lives saved through better pilot performance in actual combat.

Dr. Earl Alluisi, the former Chief Scientist at the Air Force Human Resources Laboratory, presented a notional Combat Mission Trainer (CMT) impact analysis based on data from:

- The U.S. Army Air Corp and German Luftwaffe during World War II
- U.S. allies such as Israel during the Israeli 6-day war
- Data from later U.S. combat engagements.

Dr Alluisi stated that

> All the data show that the major losses in air combat occur during the aircrews' first 8 to 10 missions. This occurs even though we know it and therefore send the new aircrews on the easiest and least dangerous missions at the beginning of their combat tours.
>
> It seems safe to infer that the surviving aircrews have learned something during those first 8 to 10 missions. Suppose we could train all aircrews in the "something," whatever it is, before they went into combat. What difference would it make?(1:36)

The implication is that if we can allow all pilots to fly their combat mission in a virtual world using a CMT, their chance of survival would be greatly

improved(4, 14, 17). The Navy has identified similar desires to let their pilots participate in one-on-one visual gunnery practice, even if the practice becomes a game, because the results have demonstrated significant improvements in gunnery accuracy when simulator trained pilots flew their first missions(8).

The Naval Air Development Center identified specific training recommendations to develop "...a "ready room available" part-task trainer..." that trains a pilot's reflex reaction to respond to a firing opportunity(8:3). The report focuses on the belief that pilots can effectively learn flying skills that are more reactive – or automatic – independent of the aircraft.

Based on the positive results of the military studies and the improved pilot survival implications, Dr. Alluisi proposed the Air Force begin improving state-of-the-art CMT's so they might grow into an *ideal* CMT. He defined the attributes of an *ideal* CMT to be low-cost and portable. The CMT must be inexpensive so every pilot could have one available and portable enough to fit in a helmet and store in a briefcase. Dr. Alluisi's vision was that such a flight simulator " ...would permit each combat pilot "to fly" his mission in the CMT-simulated environment several times before he even boarded his aircraft to fly the real mission"(1:18). He believed that to be effective, the "...experience provided for the pilots in the CMT simulation would be quite realistic relative to the actual mission that they would then fly in the real world."

Both the Naval Air Development Center and the Air Force Human Resources Laboratory clearly saw the need and advantages of a small and economical part-task flight simulator. These inexpensive systems would allow placing several in ready-rooms at the squadron level to enable pilots to practice (or play) air combat maneuvers. This practice would refine their reflexive skills and possibly improve their chances of survival.

The Air Force has since embarked on long term technology developments to create an *ideal* Combat Mission Trainer. Dr Alluisi had presented a CMT devel-

opmental framework where he defined a three pronged approach. The three major components of the *ideal* CMT are a data base collection system, an image generation system, and a crew interface/display system.

The Human Resources Laboratory has pursued these objectives and developed a good supporting system for the first two in the Advanced Simulator for Pilot Training (ASPT)(14). Many organizations are developing high resolution head-mounted displays to try and satisfy the crew interface/display system requirements(11, 18, 48). Unfortunately, the Air Force Institute of Technology's second generation HMD(7:45) does not have sufficient resolution to improve the crew interface/display beyond existing thresholds.

However, the Human Resources Laboratory recognizes that flight simulators are very expensive. The Laboratory is now looking at lesser fidelity display systems and investigating cheaper simulator systems. In addition, they are pursuing the possibility of developing a "... pilot-centered *network of low-cost, highly-specialized* simulators designed to train for large-scale combat missions"(14)[1] This lower cost approach is the essence of the part-task simulator developed in this thesis effort.

2.1.2 Research & Development. The major impact of an economical full-vision simulator is in significantly lowering the cost of designing new aircraft or aircraft components. There is little argument that "engineering simulators have allowed the military services and industry to cut costs, risks and development times on new aircraft and other systems"(19:44). A writer for the Defense Daily identifies the significant impact simulation has to Research & Development:

> As a testament to the value of simulation, every major U.S. aircraft manufacturer has constructed expensive computer simulation facilities in order to design and test, in a real-time environment, how such things as flight controls, guidance systems, flight and battle management and

[1]The large-scale combat mission is the many-on-many air combat engagement scenario.

mission avionics components would operate in an aircraft engaged in high-stress combat maneuvers.(19)

One major message is clear, as budget pressures intensify, more and more aircraft development agencies are finding the cost of full-dome simulators prohibitive and are looking for other alternatives. Air Force Lt. Gen. Michael Loh, recent Commander of the Aeronautical Systems Division, qualified that the issues of cost have become more of a limiting factor than issues of simulator fidelity(19:50).

Mr. Ken Daida, simulation director at Northrop, envisions that "as a way around the increasing costs of large, fixed simulation facilities" he could see the advancement of technology such that simulation information is presented "...on the visor of a helmet instead of in a 20-foot dome"(19:50).

2.2 Survey of Head Mounted Displays

Although we are not developing a HMD as part of this thesis, the advancement of HMD technology is directly related to the fidelity of the simulation we are able to present to a user. Therefore, this literature review includes a *survey* of HMD technology developments. This review is intended to inform the reader of the state of development of HMDs and not to provide recommendations or conclusions about the advantages of any one design.

Head mounted displays can be characterized by the number of displays used and how they present the images. There are three basic types; monocular, biocular, and stereoscopic or binocular(48:5).

monocular These HMDs use one screen to present the image to the user. This screen can be a sight or TV screen mounted in front of one eye.

biocular These HMDs display an image to both eyes. The viewing device can consist of one large screen mounted in front of both the user's eyes. An alternative design is to use two screens, one in front of each eye, and show an identical

image to each eye. The visual effect is the same as the single screen design. This is the most predominate design in HMDs today, having been popularized by VPL Research Inc..

binocular These HMDs use two screens, one in front of each eye, with a slightly offset image presented to each eye to give a stereoscopic view(33:53). Although the HMD design is similar to the two screen biocular HMDs, the supporting circuitry and software needed to maintain both images in focus and synchronized is complex.

2.2.1 HMD Categories. HMD developments can be categorized into two basic types; see-through and enclosed.

See-Through. Ivan Sutherland developed the first HMD at Harvard University as a see-through virtual world interface device(44:757). This HMD used a pair of miniature CRTs that presented a virtual image that looked to be about eighteen inches in front of the user's eyes. These images were superimposed over the real world objects that a user could normally see before putting on the viewing device. J.C. Chung presents an excellent overview of advancements in see-through technology in their article "Exploring virtual worlds with head-mounted Displays"(7:42)

Enclosed. NASA Ames Research Center developed the first fully enclosed HMD. Enclosed HMDs replace all items normally seen by a user with computer generated imagery. This type of design precludes superimposing computer generated imagery over a user's real-world surroundings(7:44). The enclosed designs can be grouped into three common types:

- The most common enclosed HMD is a biocular design using liquid crystal displays (LCDs). LCDs are fairly low resolution devices that can only support low fidelity virtual world interfaces.

- Biocular or Binocular miniature CRT designs are next. These designs use miniature CRTs to present the image to the user. The CRTs are normally mounted above the head and the image is sent through prisms and mirrors for user viewing. These designs normally use black and white or green displays, but one current research effort to create a color design is being developed at the Air Force Institute of Technology.
- Fiber Optic HMD (FOHMD) designs. The FOHMDs use a fiber optic bundle to carry a high quality video image to the HMD viewing system. The pixel resolution is a function of the number of fiber optic strands in the bundle. Success in FOHMD has been achieved(18, 48); however, the designs are expensive. Pixel drop-out is often a problem in these designs when a strand breaks. A method has been developed that lessens the negative visual effect of these pixel drop-outs and of the fiber structure itself by using prisms at each end of the bundle(18:266).

2.2.2 Related Work. A number of HMD developments have been accomplished in both academia and industry. A review of the available literature is presented below:

Frederick Brooks presents a good introduction to the virtual world concept and how to approach developing an interface to the virtual world(5, 6). He includes observations on 3-D interfaces and virtual world applications that have been researched at the University of North Carolina at Chapel Hill over the past two decades. He also provides some good insights into the general use of graphics.

The Air Force Institute of Technology's Virtual World Laboratory has developed two generations of biocular display systems based on Liquid Crystal Displays and is currently trying to improve screen resolution by developing a third generation helmet based on miniature CRTs. Capt R. Rebo developed the AFIT's first generation HMD (HMD-I) as a color biocular display system(38). Capt R. Filer im-

proved the HMD-I display in a redesigned HMD (HMD-II) that improved the image quality, reduced the total system weight, and made the head mount more adjustable and easier to wear(11). He also developed a 3D virtual environment software library to support the HMD-II, joystick, CIS Dimension Six Force-Torque Ball, VPL Dataglove, and a Polhemus 3-Space Tracker. *Maj P. Amburn* and *Lt Col J. Mills* are now attempting to improve the screen resolution of previous designs by using CRTs instead of LCD displays. Their design basically consists of three monochrome CRTs mounted above the wearers head, each with a different colored phosphor. The CRTs are aligned so that the individual images will be blended to create a full color image projected to the user's eyes through mirrors and lenses.

R. Woodruff, D. Hubbard, and A. Shaw compare five different HMD configurations(48). The comparison was done in a flight simulator experiment to answer questions about the effects of three different design criterion. Summarizing their results; 1) the closeness of optics to the eyes did not adversely affect pilot performance; 2) the boundary of a limited FOV display did not provide useful visual cues; and 3) stereopsis did provide useful visual cues to pilots when flying at close distances to other aircraft – like a tanker during refueling. The binocular view did not contribute significant cues for flying tasks that didn't require close formation flying.

Capt Caroline Hanson presents a FOHMD design using an Area of Interest (AOI) computer graphics generation method(18). AOI CIG systems provide higher resolution and brightness, and more scene detail than conventional systems because of the narrower FOV that is required for the high resolution inset. AOI is a CIG method however, and not related to the physical design of the helmet. Capt Hanson explains the design of a breadboard FOHMD system under sponsorship of the Operations Training Division at Williams AFB.

Stephen Martin and Richard Hutchinson describe a design approach for a stereoscopic HMD using miniature CRTs(33). They present a good set of HMD design considerations with a description of each. Their primary design considera-

tions are field-of-view, resolution, binocular symmetries, exit pupil size, eye relief, and eye accommodation. They present their approach to design and construction of two different helmets. They also present useful comparisons of the characteristics of miniature image sources and telescopic eyepieces.

Scott Fisher and others describe the *Virtual Environment Display System*, a head-mounted, wide-angle, stereoscopic display system developed at the NASA Ames Research Center(12). The unique feature of this system is that the virtual interface includes voice and gesture interfaces and tactile interaction through flex-sensing devices. The applications focused on telerobotics and telepresence control; workstations for management of large-scale integrated information systems; and human factors research.

2.3 Equipment Architecture

The primary focus of this research was to host a virtual world interactive flight simulator on a PC to investigate the capabilities of a low cost approach to part-task training systems. To accommodate the demanding requirements of creating real-time or near real-time simulator images that consist of hundreds of polygonal descriptions, a special graphics coprocessor was required. This coprocessor would be interfaced to an 8 MHZ Industry Standard Architecture (ISA) PC bus. Several graphics coprocessor options were investigated. The options were categorized by the type of host processor used to allow speed and feature comparisons of the coprocessor boards. This section contains a market survey of the different coprocessor boards and their features. Suitability analysis of each candidate board for use in this project is presented in Chapter 4.

One major criteria used in evaluating the boards considered in this investigation was whether a frame buffer providing NTSC signal output was available. The AFIT virtual world environment only accepts RS-170A signals. Other critical criteria included 1) programming language support; this project was being implemented

in C, 2) program development environment, and 3) compatibility with the methods being used to develop the flight simulator code in the cooperative thesis effort.

The results are summarized below:

2.3.1 Texas Instruments 34010. The TMS34010 Graphics System Processor is a 32-bit internal, 16-bit bus general purpose processor with graphics extensions. The fastest processor cycle time is 130 ns. The 34010 off-loads the burden of screen draws from the host CPU. Dramatic graphics performance improvements over standard 16-bit VGA interfaces have been achieved in supporting CAD/CAE and desktop publishing applications. The redraw time for complex images with clipping and shading are still on the order of seconds(2).

The TMS34010 offers an assortment of graphic specific instructions and special operations(46). The graphic specific instructions include single pixel manipulations, line drawing instructions, pixel array manipulation (including pixel block copies), and auxiliary instructions for clipping, trapezoid fills, and pixel evaluators. The operation specific capabilities include windowing functions, logical operators for pixel processing, transparency support, and plane masking.

2.3.2 Texas Instruments 34020. The TMS34020 Graphics System Processor is a 32-bit internal, 32-bit bus general purpose microprocessor, optimized for graphic display systems(25). This processor is a second generation graphics processor that TI claims will provide up to 50 times faster performance over their first generation TMS34010. The TMS34020 is object code upward compatible with the 34010.

A closely coupled floating-point processor, the TMS34082, operates at up to 40 million floating-point operations per second (MFLOPS) and provides floating point support to the 34020. The 34082 implements a number of specific 2D and 3D graphics math operations such as 3x3 convolution, 4x4 matrix operations, polygon clipping, backface testing, and viewport scaling and conversion. The floating point

unit operations provide a variety of instructions suitable to support real time graphic applications.

Like the 34010, the 34020 also offers an assortment of graphic specific instructions and special operations(45).

Although the 34020 has an impressive list of graphics support features, the 34082 FPU wasn't scheduled for release until March of 1990. Third party boards offering PC ISA compatible graphics systems were not expected until much later.

2.3.3 INMOS T414 transputers. The primary concept behind a transputer based system is to improve processing performance through a concurrent processing architecture(23:4). This increases the number of instructions/cycle that can be executed in any one cycle. INMOS states that the transputer directly implements the process model of computing where "a process is an independent computation, with its own program and data, which can communicate with other processes executing at the same time. Communication is by message passing, using explicitly defined channels"(23:6).

C is available as a product add-on for programming the transputers. However, the proprietary INMOS language occam is required to use as a harness to parallelize the independent C modules into concurrently running processes. A single separately compiled C program executes as a single process in an environment of occam channels(23:76).

The system configuration needed for a graphics intensive PC application would consist of the IMS B004-2 PC add-in card with an external IMS B201-1 Rack holding an IMS B007 graphics evaluation board.

The B004-2 is based on a single T414 transputer chip. The IMS T414 integrates a 32-bit RISC processor, four standard transputer communications links, 2K bytes of on-chip RAM, memory interface and peripheral interfacing on a single chip(24:30). The processor provides direct support for the occam model of concurrency allowing

the processor to share time between any number of concurrent processes. The PC add-in board comes with 2 Mbyte dynamic RAM and an IMS C002 link adaptor for connecting to other transputer boards.

The B007-1 transputer graphics evaluation board is a member of the family of special purpose transputer evaluation boards provided by INMOS. The B007-1 is also a T414 based system with a 512 by 512 frame buffer that can provide RGB output to any color monitor with scan line frequencies between 20 and 50 KHz(23:106). Occam support for this board consists of low-level graphics primitives for line and polygon drawing and alphanumeric text generation.

2.3.4 Intel 80860. Where the transputer approach achieves its performance gains through concurrency, the Intel i860 RISC processor achieves its gains through pipelining with an architecture similar to that of the CRAY I supercomputer(30). The i860 is made up of a RISC integer unit, separate multiply and add channels in the floating point unit, and a graphics unit (see figure 2.1). The chip contains several

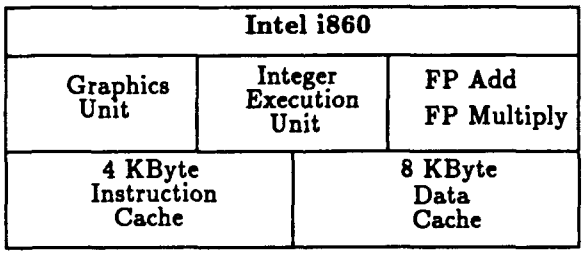


Figure 2.1. Intel i860 Block Diagram

wide information paths including a 64-bit external data bus, 32-bit external address bus, 128-bit on-chip data bus from the data cache, three 64-bit on-chip data buses for floating-point operands and a 64-bit on-chip instruction bus(15:86-87). The i860 has a 1-Kbit instruction cache and 8-Kbyte data cache in each chip die. The pipelined architecture allow simultaneous processing of a new single-precision integer operation and floating point operation at each clock cycle. "The chip outperforms compet-

ing general-purpose solutions, such as MIPS Computer's R3000, Sun Microsystems' SPARC, and Motorola's 88000 by about 50 percent"(42:33). A 33 MHz i860 can be used as a front-end processor for a display system capable of calculating 40,000 Flat shaded polygons/second and 300,000 transformed vectors/second. By contrast, chips competing with the i860 typically compute 20,000 shaded polygons/second and 100,000 to 150,000 transformed vectors/second(42:34).

2.4 Summary

There is a need for a low-cost full-vision part-task flight simulator to support pilot training and new aircraft or aircraft component research and development. A very low-cost part-task trainer could be put in every squadron operations building(s) and networked with other squadron's systems for pilots to practice combat flying maneuvers. There is *great* potential for large dollar savings in both the training and research & development worlds.

HMDs are characterized by their viewing interface (monocular, biocular, or binocular) and categorized by their type (see-through or enclosed). The HMDs of interest for our research are biocular enclosed helmets.

Powerful single chip graphics processors are coming of age. With the introduction of the Intel i860 and the TI TMS34020/34082, significant advancements in table top graphics computing are achievable. Computer image generation capabilities will continue to improve with the technological advancements in the Very Large Scale Integrated (VLSI) chip technologies.

III. System Requirements

This chapter presents the requirements for the design and implementation of the interactive virtual world flight simulator.

The basic requirement was to design and implement an interactive virtual world flight simulator on a PC that could be connected to a network to allow a two-ship simulator. The implementation of the flight simulator would follow a standard Air Force software development approach. More specifically, the code would be written in a high order language and use a classical 3D graphics reference model. Follow-on studies would then determine whether a head-mounted display, coupled with a PC, would provide a suitable platform for part-task flight training.

The high order language and graphical reference model requirements were established to decouple the software approach from any single computer architecture. This approach knowingly precludes taking a commercial flight simulator written in assembly language – specifically for the PC, and using it as the foundation for this research. From an Air Force perspective, the software developed for a portable part-task trainer would be expected to have a life-cycle of 15 to 20 years. The improvements in desk-top computing technology is advancing a new generation every five years. Software portability and the ability to support and enhance the software using military resources are crucial aspects of any continuing research in software for military systems.

The motivation for focusing on the portable software issue stems from the high cost of software development and computing resources. Software designed and developed for a particular platform costs more than software that has been previously written and tested, then ported to other platforms. If machine dependent code is allowed in the development of a portable part-task trainer, the Air Force will find themselves unable to change computing platforms without incurring exorbitant

cost to redevelop the software. Other advantages of software reuse have been well documented(3, 22, 41) and include many of the quality factors the Department of Defense has been concerned about since the *software crisis* began(3:3). The most viable Air Force approach to the problem is to have portable software, and use it across new generations of computing hardware over the life-cycle of the software.

3.1 General User Requirements

The AFIT virtual environment display system (VEDS)(11) would be used for the virtual world interface. VEDS provided the hardware and software foundation for the HMD and Polhemus tracking system. The institute's HMD consists of two Sony FDL-330 color TVs. The FDL-330 TV is a three component monitor system with separate detachable power pack, tuner, and monitor sections(11). The screens are 2.7 inches diagonal with a 360 $\times$ 240 pixel grid. Since this was a color system, 3 pixels were combined to represent one color element for an effective resolution of 120 lines. The two TVs were butted together and mounted approximately 6 inches in front of the viewers eyes. To accommodate the optical distortion from such a close viewing range, a set of LEEP optics (designed by Eric Howlett, built by Pop-Optix Labs) are used. These are the same optics used by NASA Ames in their virtual environment display systems(12). The Sony monitors are too large to align with the central axis of the LEEP optics so Fresnel press-on prisms bend the light from the monitors to direct the light directly into the eyes as shown in Figure 3.1. The Polhemus 3Space Isotrak magnetic tracking device consists of two sensors, one that remains stationary and one that is attached to the viewer's head. The sensors consist of three orthogonal magnetic coils. The stationary sensor transmits electromagnetic pulses from each coil in turn. The signal strength is then sampled in the sensor on the viewer's head. The resultant signal strength at the receiving coils is used to determine the exact position and orientation of the user's head(9, 35).

The virtual laboratory equipment would be interconnected over a thick wire

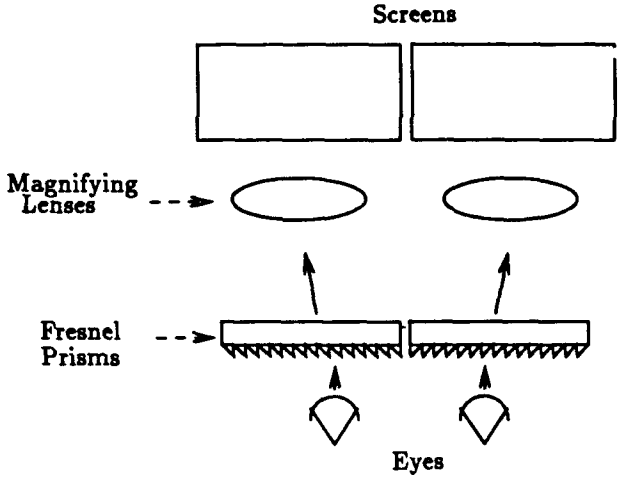


Figure 3.1. AFIT HMD II Design

ethernet running TCP-IP (Transmission Control Protocol - Internet Protocol). This configuration dictated the basic network interconnect required for this flight simulator.

Flight training requires a rapid picture update rate for real-time simulation. A rate of 30 frames per second provides the appearance of smooth motion in an animated sequence. Although an update rate of 30 frames/second is desirable, a lesser rate would still support our investigation; and an update rate of 15 frames per second was established as a goal.

This update rate requires a rapid graphics update capability from the host computer system. The PC is based upon a general purpose processor with no enhancements or optimization for graphics. Additionally, RS-170A (or NTSC) output is required for the institute's virtual environment; therefore, the PC's standard EGA or VGA capability was insufficient to meet the requirement and would have to be supplemented with a graphics coprocessing system.

3.2 Specific Requirements

A specific requirement of the project was to modify an existing flight simulator hosted on a conventional 2D screen display with a keyboard and mouse interface to the virtual environment without any loss of performance capabilities. This generated additional user interface requirements for displaying the cockpit controls and out-the-window scenery as well as a new user control interface – the keyboard was not an effective interface for a user wearing the enclosed display system. The new display interfaces were required because the screen resolution of the AFIT HMD-II is insufficient for displaying cockpit controls on one third of the display screen.

3.2.1 User Interface. The primary user interface for controlling the flight simulator is through the joysticks. The aircraft functions that required interfacing to the joystick included:

- Steering
- Throttle control
- Firing guns
- Firing missiles
- Firing rockets
- Rudder Control
- Spoiler Control
- Landing Gear Up and Down
- User Menu Activation and Selection
  - Wingman's View
  - Tower View
  - Restart Game

3.2.2 Software. Three basic requirements for the software used on the platform were established.

First, the UNIX operating system would be used rather than DOS or OS/2. This decision was not based on any perceived deficiencies with the other options; rather it was made to maintain compatibility with the operating system being used in the cooperative thesis effort. The vast UNIX experience available at the institute was also an influencing factor.

Second, the flight simulator code chosen as the starting point for this thesis had to be compatible with the SGI 4D/85 GT (the other computer system purchased for the cooperative thesis effort) and written in the C programming language. The specific software language requirement was established because of the availability of software development tools. Each vendor's UNIX operating system provides C software development environments at little or no extra cost.

The major software development requirement established was to rehost the flight simulator to the PC with minimal loss in functional capability. This requirement spawned the need to develop a functions interface library (emulation) from the Silicon Graphics graphics library to the PC graphics engine. The flight simulator could then be rehosted to the PC with little modification.

3.2.3 Hardware. The use of a low-cost CIG (Personal Computer) in contrast to a mid-cost CIG (SGI 4D/85 GT) being used in the other half of the cooperative thesis was the basis for the hardware platform selected. A fully functional flight simulator executing on both machines could then serve as a common foundation for system performance evaluations.

The specific UNIX operating system selected (ESIX System V – see appendix D) for the PC required the use of specific brands of equipment. UNIX device drivers had to be available to drive all peripheral devices contained in the computer system. These devices included major elements such as the ethernet card, VGA card, serial

and parallel ports, and any other device using the back-plane bus. The subsequent equipment selections were based upon this operating system support. In addition to the restrictions imposed by the available device drivers, ESIX System V required a minimum of 4 MBytes of RAM and a 40 MByte hard disk.

The final system configuration was defined to be an 80386 PC platform with:

- 100 Megabyte or larger Hard Disk
- 5 Megabytes of memory
- 80387 floating point processor
- Graphics engine providing NTSC output
- Standard VGA driver for software development
- A VGA monitor
- An NTSC color monitor
- Ethernet card
- Multi-IO card with serial and parallel interfaces
- Joystick card
- 2 joysticks

3.3 Summary

Requirements were established to port an existing flight simulator program to a PC platform. The PC would include a graphics processing unit and would be running the UNIX operating system. An emulation of the Silicon Graphics IRIS 4D/85 graphics library would be developed to host the flight simulator code on the new machine. A new user interface would be designed for the flight simulator because a user wearing the HMD could not see the keyboard to adjust flight parameters.

IV. System Design and Implementation

This chapter covers the theory and rationale of design decisions made during the development of the system. General hardware design considerations based on the graphics engine selection are presented first. Following this, the basic theory behind the Programmer's Hierarchical Interactive Graphics System PLUS (*Plus Lumière Und Surfaces*) (PHIGS PLUS or PHIGS+) graphical reference model is presented. The software design approach and functional partitioning are then addressed. Implementation problems that required design changes complete the system design description.

4.1 Hardware

The computing platform for this thesis consisted of a Compaq 386/20[1], with an 80387 floating point coprocessor, 5 MBytes of memory, a 120 MByte hard disk, VGA card and monitor, one serial port, and two game ports with joysticks. To support the graphics update requirements and the RS-170A output requirement, the basic PC configuration was enhanced with a Real World Graphics Ltd. PC Reality board using two Intel i860 RISC processors. An additional 19 inch NTSC compatible monitor was connected to the PC Reality board through SMB connectors on the back rib of the add-in card. This facilitated the use of the 19 inch NTSC monitor for the graphics output and the VGA monitor for development and debug.

The selection of the graphics add-in card merits some discussion to provide specific conclusions reached during the graphics engine investigation.

4.1.1 Graphics Add-In Card Selection. As identified in chapter 2, several capable processors exist that can be connected to an ISA (Industry Standard Architecture) PC-AT bus. The objective was to choose an ISA compatible card that

[1]Operating at 20 MHz.

provided the RS-170A signals for the AFIT VEDS system while achieving the 15 frame/second update rate.

For RS-170A signal support there are various design solutions ranging from a VGA-to-NTSC encoder board to the special purpose graphics processors.

The VGA-to-NTSC encoders available(16) offer graphics update rates determined by how fast the TMS34010 (on which the boards are based) can draw the screen. VGA graphics monitors have suitable bandwidth to support real-time graphics required for flight simulation. In fact the VGA standard horizontal scanning frequency is 31.47 KHz, double the 15.734KHz of an NTSC monitor. The bottleneck lies in the ability of the processors in the PC to generate the image into the frame buffer. Therefore, the standard PC using a VGA-to-NTSC encoder would not provide screen updates fast enough to meet the 15 frames per second goal.

The alternative to the VGA-to-NTSC approach was to select a special purpose graphics add-in card to speed the graphics display. There were several candidate chip sets that could serve as the heart of a special purpose graphics card. The following discussion provides the rationale for the selection or non-selection of a particular chip set or add-in card.

4.1.1.1 TI TMS34010. The TMS34010 general purpose processor with graphics enhancements has become the heart of a number of graphics add-in boards that are supporting Computer Aided Design (CAD) processing systems. The 34010 based boards provide approximately 5 times the performance of existing 16-bit VGA systems(46:45). This performance improvement is a significant enhancement to 2D graphics applications, but is not sufficient for 3D graphics in real-time.

4.1.1.2 TI TMS34020. The TMS34020 was TI's answer to the 3D graphics performance question. The TMS34020 coupled with the TMS34082 FPU should

provide sufficient speed to support real-time 3D graphics[2]. The enhanced set of graphics primitives built into the chips command set provide a rich graphics environment from which to work. Coupling the capabilities of this chip set with the TMS34010 Graphics/Math Library provides a complete 3D graphics application system. Unfortunately, the TMS34082 FPU was not projected to have production units produced until 2nd Quarter 1990. Third Party development of graphics add-in boards would lag behind that by about 6 months. This time schedule prevented the TI TMS34020 from further consideration because the boards would not be ready in time for this thesis effort.

4.1.1.3 INMOS Transputer. The Transputer technology offered additional speed because of its concurrent approach, but it also added a considerable amount of complexity to the problem. Instead of just implementing a virtual flight simulator, the problem domain would be expanded to include concurrent processing hazards. Concurrent processing would force considerable restructuring of any applications programs or libraries planned for use as the foundation for this development. Transputers may provide a viable approach to the problem, but at a higher complexity level and cost than other architecture alternatives.

4.1.1.4 Intel 80860. Intel Corporation's new RISC processor – the i860 looked like the answer to the graphics performance problem. Third party boards were just entering the market and advertised impressive performance capabilities. I elected to focus on the i860 based graphics boards available on the market in March 1990. There were three candidates, 1) Hauppage Computer Works, Inc. with an 80486 motherboard that accepted an i860 as a coprocessor, 2) ALACRON Inc. that

[2]One implementation of a 34020 based system was demonstrated at SIGGRAPH '90. The board was the Daewoo Graphics & Imaging System with one 34020 and four 34082 (one to process each quarter of the screen). This board was not capable of generating real-time graphics images at 30 frames per second. The vendor stated that all images shown at the conference were recorded animation sequences and that the board was not capable of real time animation. Further review of other vendors products is recommended before considering this board.

offered an i860 add-in card, 3) and Simulation Technologies who marketed an i860 add-in card for a U.K. firm – Real World Graphics Ltd.. I will explain the rationale used to choose between the three.

Hauppage. The 80486 motherboard replacement approach looked appealing. Hauppage advertised that a daughtercard frame buffer was available that would be able to provide RS-170A signal output. After contacting Hauppage, I discovered that the frame buffer was only in the planning stages and the marketing people I spoke with had no idea when development would begin on this daughtercard. Without an RS-170A output capability, the board would not help. Using a VGA-to-NTSC encoder would slow down the graphics pipeline because of the slower TMS34010. This approach was abandoned.

ALACRON. ALACRON offered an i860 add-in board compatible with the ISA AT bus architecture. Their board consisted of one i860 but had no frame buffer. The other major set-back for this approach was that the application program would have to be developed from scratch to work on the i860. Algorithms would have to be parallelized to take advantage of the chips separate integer and floating point Arithmetic Logic Unit (ALU) pipelines. Another major set-back was the cost of the Intel i860 development kit. The ALACRON board was reasonably priced, but the development kit put us over budget.

Real World Graphics. The PC Reality board was Real World Graphic's entry into the i860 market. They had been working on implementing a single board CIG for some time, first with transputer technology, then with the Motorola 88000 microprocessor, and finally migrated to the Intel i860 upon its release. A significant advantage was expected to be from the PHIGS+ library provided with the board. The application programmer could not develop any native i860 code and was restricted to interfacing with the i860 through the PC Reality PHIGS+ library

calls. A variety of output interfaces from the on-board frame buffer were available through specification to manufacturer, including the required RS-170A signals.

The graphics support reported by Simulation Technologies included:

- Provide rendering features that include Gouraud and Phong shading, antialiasing, shadowing, and texturing.
- Perform geometry and rendering algorithms in standard programming languages through the PHIGS+ graphics library.
- Offer improved performance through parallelism while hiding the parallelism from the applications program. This approach would allow sequential programs to run on the system without modification when additional processors were added.
- Provide up to 10,000 Z-buffered polygons at 60 frames/second. The written literature provided no claim as to whether this rate is for flat-shaded or Gouraud-shaded polygons. Mr. George Keverian, vice-president of Simulation Technologies – the U.S. subsidiary for selling the PC Reality boards, clarified that the claim is for Gouraud-shaded polygons.

Real World Graphics has not yet met their performance goal. The S. Klein Newsletter on Computer Graphics reports a fully configured system handles 6600 Gouraud shaded polygons at 30 frames/second(29).

The PC Reality board architecture is shown in Figure 4.1. The board contains two Intel i860 processors, one acts as a preprocessor to the other. Standard configurations contain 4 or 16 MBytes of on-board RAM with a 1 MByte frame buffer memory. We opted for the lower priced 4 MByte option. The more memory available, the more object structures could be stored in memory. The frame buffer is configured as a 1K x 1K x 24 bit buffer with an additional 4 bits of overlay. The displayable resolutions are configured by the manufacturer and include NTSC 640

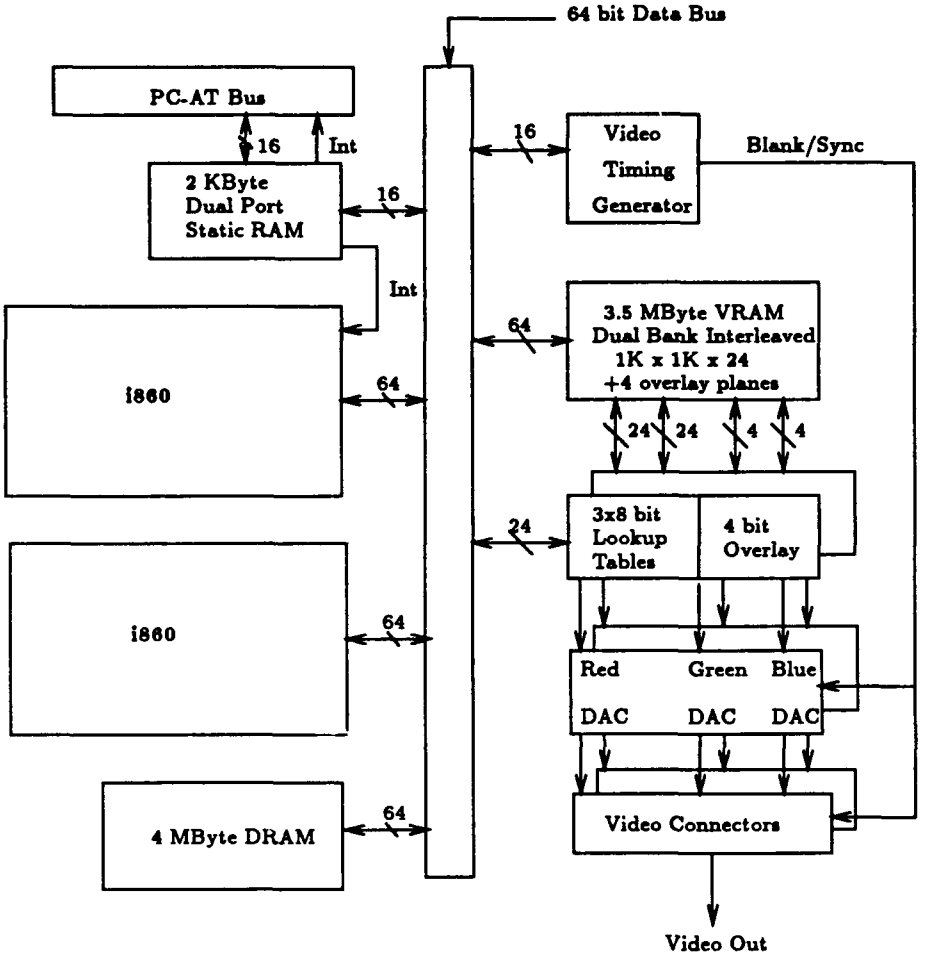


Figure 4.1. PC Reality Processor Block Diagram

x 484 or PAL 768 x 576 interlaced, and up to 1024 x 768 non-interlaced resolution. With the board configured for NTSC output, double buffering is the only method of screen draw. Double buffering is the process of displaying one image from half the frame buffer memory while drawing the next image to be displayed in the other half of frame buffer memory, then swapping the newly drawn picture to the display. The board has a 24-bit color look-up table (8 bits per color) enabling true color display of 16.7 million colors.

4.1.2 Final Graphics Board Selection. The three best candidates of those considered were the TI TMS34020/34080, Intel i860, and Transputer based systems. The PC Reality Board based on the Intel 80860 processors was chosen because it offered a system that met all the initial developmental requirements needed for the prototype simulator.

The PC Reality represented a state-of-the-art graphics coprocessing system that could provide the required RS-170A (NTSC) output signals. The board employed a pipelined architecture (similar to the CRAY I) that appeared to provide enough power, in terms of drawing Gouraud-shaded polygons, to meet the real (or near-real) time graphics requirements. The company had also advertised a robust graphics library that was provided as the programmer's API. This meant that no i860 code would have to be developed, nor would i860 to 80386 processor synchronization or message passing be necessary. The concurrent processing problem was not part of the developer's responsibility.

The TMS34020 based systems were not selected because of the non-availability of the 34080 coprocessor. Development boards based on this architecture simply weren't available at the start of this prototype development.

Transputer systems offer another possible solution to obtaining the real-time processing power needed; however, they also require the programmer to work within the concurrent processing paradigm. This is a more difficult environment then the serial or pipelined architectures used in the sequential paradigms. Transputers had a C language programming environment;however, each C program runs as a single thread (sequential) program. To attain the concurrent software structure, each C program must be tied together with the others through a concurrent harness using the occam programming language. Since the plan was to port an existing flight simulation program to the PC, the use of transputer technology meant that the sequential programming model that the Silicon Graphic's Dog program used would have to be changed to a concurrent model. This added complexity that was not

desirable for this prototype development.

4.2 Software

The PC Reality Board is supplied with a graphics library that provides the Application Programmer interface (API) between the application programmer and the underlying Reality Graphics Environment (RGE) functions implemented in i860 code. This graphics library interface provided benefits and disadvantages for the application programmer, some of these will be discussed below.

The Reality API is modeled around the PHIGS+ standard and shares many, *but not all*, of its features. The Reality library provides the application programmer with functions for the building and displaying of 3D objects from basic graphical primitives. Since the board was very new to the market (serial number 19) the actual graphics library was still being developed by Real World Graphics with a new library release occurring as late as November 1990, much too late to be used for this project. This late release contained the 2D graphics functions and *text* functions that were defined in the PHIGS standard and included in the developer's handbook(36)[3]. This late library release was to provide functions needed to use the advertised 4-bit overlay plane. These missing functions prevented use of the 2D graphical function calls used in Dog which spawned a new requirement to develop a cockpit using 3D functions.

PHIGS is an International Standards Organization standard describing a graphical reference model that provides functions for application modeling and three-dimensional interactive computer graphics(21, 20). This standard has descended from the CORE and GKS standards and it's functions closely resemble functions from those graphics standards. The Real World Graphics implementation of the PHIGS+ library did not implement all the functions defined by the PHIGS standard(26),

[3]The absence of the needed 2D and text functions was not known during the selection process presented above.

those functions that were essential to this application but not implemented are identified.

4.2.1 PHIGS PLUS Reference Model. PHIGS defines methods for geometry definition, display, and editing; PHIGS PLUS added higher quality 3D picture rendering techniques (lighting, color and shading) and more sophisticated geometries (tesselated surfaces)[4].

All objects are hierarchically modeled into a static structure called the Central Structure Store (CSS). The objects are defined in a modeling coordinate system that is convenient to use to describe the object. Modeling transforms can then be applied to move the components into proper position thus combining the different parts of the picture together into a single world coordinate (WC) system.

There are two levels of modeling transforms: local and global. The combination of these is applied to all object coordinates in the structure, with the local transform being applied first. The composite modeling transform (global $\times$ local) is inherited from parent structures allowing complex hierarchical transformations to be constructed (21:37).

There are three levels of data structuring:

- Structure elements
- Structures
- Structure networks

Structure elements are the basic element of the modeling system. Each structure element can be an output primitive (geometry definition), attribute specification (shading parameters, etc.), modeling transform (local and global), view index (to

[4]Reference (21) provides an excellent tutorial on the PHIGS and PHIGS PLUS graphical reference models. Readers desiring a more detailed description of the model should consult (20) for ordering instructions.

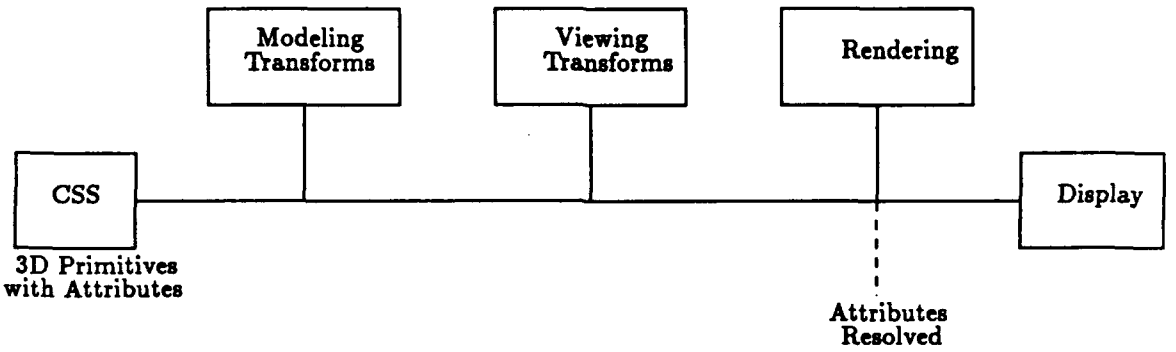


Figure 4.2. PHIGS Graphics Pipeline

change views), label (marks a position in a structure), name set (defining visibility and highlights such as on a menu), pick identifier (e.g. picking a menu selection), or an execute structure (executes another structure providing the method for arranging structures into hierarchies).

A structure is a sequence of structure elements. Various structure definitions can be combined into a single parent structure creating a structure network.

Structures can be edited changing any of the structure elements. The use of labels is the easiest method of referencing into a structure by simply setting the structure pointer to the name of the label [5]. By editing the local or global transformation elements, the objects defined in the static structure store can be moved.

After structures have been defined they can be posted to the CSS as an active component and then drawn, which sends them through the PHIGS output pipeline (Figure 4.2 illustrates the viewing pipeline). The pipeline consists of three stages:

1. Modeling transformations – applies modeling transforms (if any) which maps from modeling coordinates to WC.

[5]This is one of the features that doesn't work in the PC Reality PHIGS library version 3.2

2. Viewing transformations

   (a) View orientation transformation – maps from WC to View Reference Coordinates (VRC).

   (b) View mapping transformation – generates the view and maps the VRC to Normalized Projection Coordinates (NPC).

3. Rendering

   (a) View clip – clip against planes of a cuboid viewing volume in NPC.

   (b) Attribute resolution – color and shading.

The view orientation and view mapping transformations are both specified as 4 × 4 homogeneous transformation matrices by the user. One note of caution, the transformation matrices are designed to be multiplied by column matrices, *not* row matrices. This ordering is the transpose of the Silicon Graphics transformation matrix.

The software developed in support of this thesis utilizes a perspective view mapping. Understanding the key components of the perspective view components under the PHIGS standard is essential to utilizing this reference model. Figure 4.3 illustrates the key components for a viewer oriented perspective view.

The perspective view contains a Perspective Reference Point (PRP) or eye point (position). Unlike the center of projection in the *Core* graphics system which specifies the center of projection in world coordinates(13:281) the PRP is specified in *view* reference coordinates (VRC). When specified at the origin (0,0,0), the PRP is colocated with the View Reference Point (VRP). Figure 4.3 shows the VRP slightly offset above the PRP only to illustrate the components. The VRP is specified in *world* coordinates to define the origin of the right-handed VRC system in world coordinate space. By changing the world coordinate position of the VRP, the origin of the viewing volume is translated in world space. For a viewer centered view (such

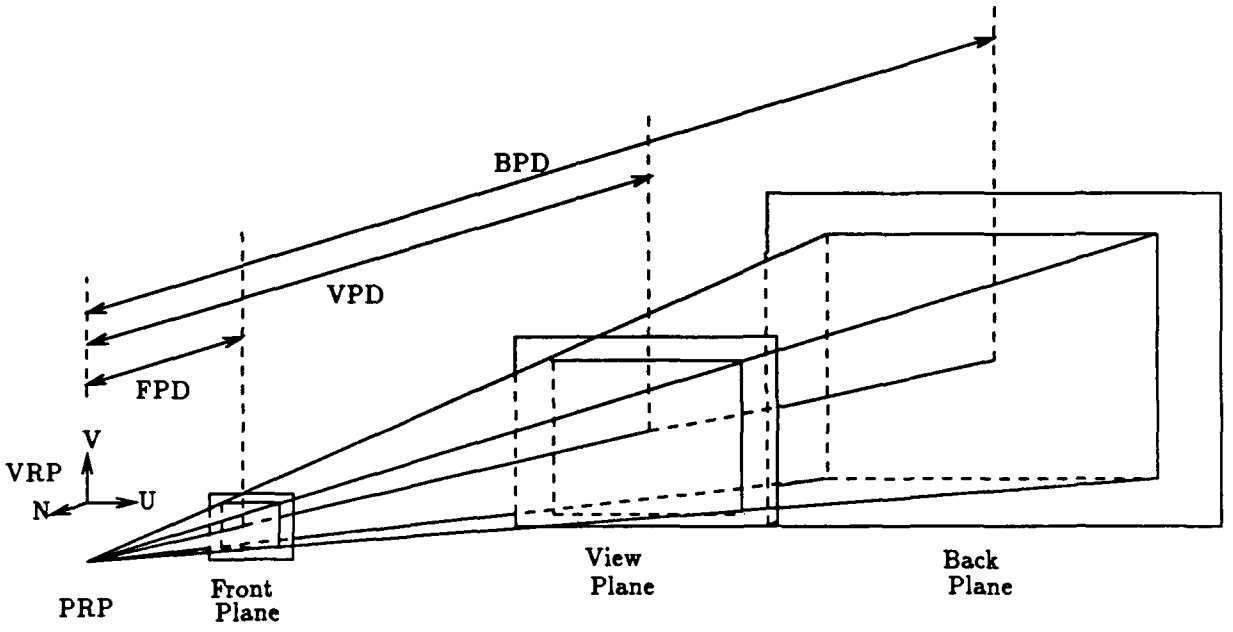


Figure 4.3. Viewer Oriented Perspective View

as that needed for a flight simulator), the PRP is defined to be at the origin of the VRC system. The view reference point can then be set equal to the aircraft (or pilot's) location to provide the correct position of the viewing volume in world space. The VRP can also be located on or near an object of interest with the PRP specified at some positive distance in z (0,0,z) for an object oriented perspective. The programmer must specify a view up vector **V** and a normal vector **N**. The positive normal points from the back plane to the PRP. The **U** vector is calculated by the PHIGS graphics pipeline by crossing the **V** and **N** vectors (**V** X **N**). The orientation of the viewing volume is specified by the definition of the **V** and **N** vectors. By rotating these vectors opposite of an aircraft's roll, pitch, and yaw, the correct orientation of the viewing volume is stipulated. The front plane defines the front of the clipping volume, the back plane defines the back, the view plane must lie between the front and back planes and represents the window into the graphics world. Note that the front, view, and back plane distances (FPD, VPD, BPD) are

measured relative to the center of the VRC system along the negative z axis and are specified in view reference coordinates. If the center of interest was between the front plane and the view plane, the FPD would be positive and the VPD and BPD negative values. Note in this case that the PRP cannot lie between the front and back planes so the PRP would have to be specified at some positive z value greater than the FPD.

The PHIGS workstation concept allows the binding of an application to the specific machine dependent methods necessary to present a picture. This concept allows the use of several different terminal types for one PHIGS implementation. The PC Reality implementation is only valid for a single workstation type and does not use this concept. The workstation identification number variable must still be passed when making function calls, but none of the library functions utilize the variable.

Different workstations also provide different graphics regeneration capabilities in a *deferral* mode. These capabilities can be exploited to give the user control over the amount of delay that occurs between making changes to the picture and the resultant output to the workstation. The PC Reality only offers one deferral mode option – ASAP (As Soon As Possible). This immediately updates the picture displayed on the workstation's screen.

4.3 Application Software Approach

The software project requirements were to implement a flight simulator on the PC that could be networked to another flight simulator running on a Silicon Graphics IRIS 4D/85 GT mini-computer. Silicon Graphics Inc. provides no fee source code for a networked flight simulator – Dog. This source code provided a foundation from which we could begin work at a reasonable application layer to integrate a Virtual World Interface.

On 23 July 1990, Mr Keith Seto, Technical Marketing Manager at Silicon Graphics, gave verbal permission to use and modify the Dog software. This included

permission to rehost the software on a PC chassis running the UNIX operating system. He promised written permission at some future date.

The Dog program is a collection of source code that has been built up from a stand-alone flight simulator implementation called Flight. The software makes extensive use of the Silicon Graphic's immediate mode graphics pipeline. In addition to including the new graphics functions released with the SGI 4D architecture, the program contains significant use of older functions that constituted the outdated SGI IRIS 31XX graphics pipeline methods. The SGI 4D machines provide built-in support for those older methods; however, they clearly identify that those methods do not optimally exploit the graphics pipeline provided in the 4D architecture.

The task at hand was to develop a software emulation library that would emulate the function calls used within the Flight and Dog programs. This included graphics functions for both the outdated IRIS 3130 and the newer 4D graphics pipelines. Many of the graphics functions exploited hardware characteristics of the SGI machines (hardware Z-buffer) and simply could not be emulated with the PCR hardware or associated API library. These functions were commented out and when the visual results were unacceptable, a work-around solution was devised.

The development started with four fairly significant handicaps:

- The PC Reality API was an incomplete implementation of the PHIGS standard.
- PHIGS includes no ability to modify items at device coordinates.
- The PC Reality board did *not* provide the required RS-170A signals.
- Real World Graphics provided marginal documentation supporting their PC Reality (PCR) board and API library.

Several features were missing in the first delivered version of the PCR library (version 3.2). The most significant was the lack of any of the documented two-

dimensional functions. The 4-bit overlay planes had no library support which prevented early development efforts to implement cockpit displays. There was also no text support (either 2D or 3D), preventing instrument panels, status displays and menus from being implemented. Several graphical primitive methods had also not been implemented. These included:

- 2D and 3D line drawing capabilities. The Dog software made extensive use of both 2D and 3D line drawing to draw grid lines and instrument displays.
- Points and markers. Again, these were needed to support the instrument displays and point light sources.
- 3D hollow polygons.

The advanced shading methods (Phong) were in the User's Manual but not implemented. The same applied to transparency and other surface property functions. The library certainly appeared to resemble a Beta test implementation rather than the high priced production quality support library we thought we had purchased.

Real World Graphics Ltd. incorrectly implemented the modeling pipeline in the RGE (i860 code). Object transformations occur through specifying a local transformation matrix and global transformation matrix. The local modeling transform is applied first followed by the global transform resulting in a composite modeling transform. The PCR library correctly implements the transformations of the local transformation matrix by transforming the objects relative to the modeling coordinate space. The *incorrect* implementation of the global transformation matrix transforms objects relative to the VRC system rather than the world coordinate system.

The PHIGS standard defines all graphics primitives at the object level. In so doing, it does not allow for methods that write directly to the screen in device coordinates. This omission from the standard prevented me from implementing or emulating the missing PHIGS functionality needed for a complete implementation

of the flight simulator. Flight uses numerous screen writes, in device coordinates, for drawing the 2D cockpit and text messages. This difference in standards may not have been a factor, but since the Reality graphics library didn't contain similar functions at the object level[6], I was unable to determine if the omission was critical. Regardless, a substitute cockpit had to be developed using the 3D function calls available in Version 3.2 of the Reality graphics library.

Real World Graphics Ltd. had specified that their board output NTSC resolution as one of the output options. The claim was clarified through the United States distributor, Simulation Technologies Inc., that the output included signal compatibility with the RS-170A standard as well as screen resolution. However, at the end of the development, I found that the signal from the Reality board was 30 Hz non-interlaced output; whereas, RS-170A is interlaced. The virtual environment was completely implemented and only needed to have the signal output in composite form (standard U.S. TV signal) to feed the signal to the LCD TVs in the AFIT HMD II. Subsequent contact with Real World Graphics Ltd. indicated they believed interlaced output was possible but they needed to research what changes were required to implement the change. The change was supposed to be a change to a firmware switch in the RGE that Real World believed could be made from the Rinit() function call in the graphics library. Real World Graphics Ltd. never replied with the needed method to switch to interlaced mode; this ended the hope of using a virtual environment with the flight simulator.

Finally, the documentation provided with the PCR board documented a number of functions that were not implemented in the PCR API library. The tutorials provided with the manual had errors and required correction before they would execute. Even the hardware installation instructions were misleading. The instructions were correct, but the board was shipped with the wrong base address set.

[6]These functions were not included in Version 3.2 of the Reality Graphics Library but have been advertised for Version 4.0.

4.4 Software Development Methodology

A functional software development methodology was adopted for this application development. Both the PCR API library and the flight simulator code were written using the classic functional approach in the C programming language. Since these two source programs were the foundation of the prototype development, the natural transition to a derivative work was to also use the functional approach.

The modular decomposition of the emulation design was already defined by the function descriptions in the SGI graphics library. The graphics library reference manual(40) provided written specifications for all the required functions. Only those functions used in the flight simulator software were emulated. These were isolated by removing the SGI graphics library from the linker and having the linker identify all the unresolved functions.

Simplifying the development methodology was important to the development approach because of the complexity of the existing program being modified. The Flight software consists of approximately 20,000 lines of C code with few comments. The code was built by numerous programmers over the years which introduced various coding styles and programming errors. Several of these errors have been encountered (and corrected) which is just a symptom that many more exist. Adding to the complexity of the approach was the immature PCR API library. Several errors encountered in the library logic prevented full use of all functions.

Every attempt was made to keep the software emulating the SGI graphics library loosely coupled by localizing variables as file global rather than program global variables. However, because of the transformation from the SGI graphical reference model to the PHIGS model, some program global variables were required to define a common set that were essential to both graphical reference models. The globals were limited to parameters defining the PHIGS view reference model, lighting attributes, object attributes, and global indexing for the PCR on-board array storage. Unfortunately, some control binding was introduced by the global indexing into the

PCR on-board arrays. These on-board arrays were used to hold modeling and viewing transforms for certain objects. The only means available to establish more than one viewport on the screen was through the use of the PCR viewport parameters array. In an effort to adhere to modern programming practices(3, 41), constructors and selectors were implemented for often used global variables[7]. By using constructors and selectors, the programmer's interface to the underlying data structure was buffered. Knowledge of the specific data structures needed by the PC Reality function calls were isolated to those few places where the actual function emulation was implemented.

Another criteria used for the design of the graphics library emulation was *changeability*. The objective of designing for changeability meant that modules were not heavily impacted by changes. This design criteria was crucial because of the size of the project. The principles of information hiding and localization of data objects was used as the method to support changeability. By hiding the data structures from the programmer, and localizing those variables that were common to just a few functions, there was little impact to the program when local methods had to be changed.

Both Kernighan and Ritchie C and ANSI (American National Standards Institute) C syntax was allowed. The Free Software Foundation's GNU[8] C compiler accepts both formats and was used as the project compiler. The advantage of this approach was that more flexibility was allowed in reusing previously developed code. A number of modules coded in ANSI format were available for reuse, without modification, from previous graphics renderers developed at the Institute. These ANSI modules were used in the emulation software and integrated with the Flight software coded in Kernighan and Ritchie syntax.

[7]Constructors are functions that alter the state of a data structure, a selector evaluates the state of a data structure or returns the value of the data without altering the state of the structure.

[8]Glad it's Not UN*X project.

4.5 Software Design Notes

Two processing platforms were available within the computer system: the 80386 general purpose processor with 80387 floating point support and the PC Reality board with two i860 processors. The PC Reality board was only available over the 8 MHz ISA bus. This architectural break-out drove a software design decision to partition the graphics emulation functions to exploit the 80386/80387 pair of processors when convenient. The basic matrix support on the Silicon Graphics machines is a matrix stack. This stack contains all the transformations that are used for transforming graphical object descriptions as well as the view and orientation transformations. The PCR library does not use a matrix stack, rather it uses an array of matrices only accessible through pointer indexing. I made the decision to design a local matrix stack rather than use the matrix storage and indexing provided on the PCR board for two reasons. First, the SGI library uses a matrix stack so by having a matrix stack emulation, the transition from the SGI graphics system to the PCR graphics system was more direct. Second, the PCR matrix storage was better utilized to store local matrices that could be modified to move objects[9].

To effectively use the PCR matrix storage capability, the matrix had to be retrieved from the PCR board, transformations applied, and then shipped back over the ISA bus to the PCR board. This looked to be a costly operation over an 8 MHz ISA bus. After designing and implementing the matrix stack, I received the source code for the PCR API. Real World Graphics had made the same partitioning decision. All transformation functions are local functions executed on the host computer's processor. There are no stated requirements from Real World Graphics for 80387 floating point processor support on the host computer. Execution of these transformations on a system containing only an 80386 would cause additional degradation in speed performance.

[9]See Appendix C for the methods used and alternatives available.

For the Reality graphics system to correctly display object color, surface properties had to be specified for all objects. The fast graphics mode for the Reality system is a Flat Shade mode. There is one documented and one undocumented side effect when using the Flat Shade mode. First, only two lights can be specified. Although the documentation doesn't prohibit making both light sources ambient, the library didn't operate correctly when so specified. One light had to be ambient and the other directional (infinite) to keep surface shading constant when the flight simulator was flown upside down. Second, the undocumented side effect was that only 16 calls to the function to set surface properties were effective in the Flat Shade mode. When the 17th structure was posted to the CSS, the surface properties didn't take effect and the object was not displayed in it's true color. This drove a work-around in the way that the Silicon Graphics object descriptions were read into the CSS. Instead of posting each independent object (structure), like a cockpit, wing, building, etc., with a unique surface property[10], a master object (structure network) had to be created for the entire aircraft with only one surface property applied to all child structures. The same approach was used to draw the mountains, runway, infinite_world, and static elements of the 3D cockpit. Unfortunately, indicators for flaps, spoilers, and landing gear were dynamic and could not be executed from the structure network and then independently unposted from the CSS. To have independent objects turn on or off as needed, they had to be posted to the CSS as a unique structure. The visual effects were undesirable in that the indicators did not display up in their true color; however, they did show up and were able to show the state of the aircraft components.

Appendix C contains additional detailed notes on how the SGI 4D library was emulated using PHIGS function calls. This appendix also describes what modifications were necessary to the flight simulator software to use this emulation library and host the software on the PC platform.

[10]This feature is needed to specify the cockpit as transparent while the wing is metallic.

4.6 Summary

The Real World Graphics Ltd. PC Reality board, using two Intel 80860 RISC processors in a pipelined architecture, was selected as the graphics engine for our "state-of-the-art" PC. The graphics API accompanying the board was based on the PHIGS PLUS standard; therefore, the library emulation had to transform the Flight software from the SGI graphical reference model to the PHIGS PLUS graphical reference model.

The PHIGS standard is the newest international graphics standard defining a 3D graphical reference model. Those parts of the reference model needed to program the flight simulation were explained.

A functional programming approach using the sequential programming paradigm was used for this project. Desirable software engineering principles were followed to try and control the complexity of the programming task.

Actual design and implementation of the emulation library and porting of the Flight software was complicated by specified, but missing functions in the early version of the PC Reality graphics library. The lack of 2D and text functions spawned a new requirement for a redesigned cockpit. Other errors in the Reality Graphical Environment (i860 code) forced additional work-arounds for representing motion and surface properties.

V. System Implementation

The Silicon Graphics' flight simulator Flight was ported to the PC; however, the resultant frame rate fell far short of the goal. This chapter describes the implemented systems capabilities and limitations, and provides conclusions and recommendations resulting from this research.

5.1 Project Goals

The project goals were to build an SGI graphics library emulation, using this emulation – port the flight simulator Dog to the PC platform with no loss in simulator functionality, build a virtual world interface to the simulator, and network the simulator to the Silicon Graphics IRIS 4D/85 GT to fly a two-ship scenario. The first three goals were achieved to the extent that the PC Reality API would support. Evaluation of the flight simulator capabilities at this stage of development showed that the host PC and PC Reality boards' capabilities to support the classic 3D graphics flight simulator were not sufficient to pursue the last goal.

5.2 Capabilities Implemented

An SGI graphics library emulation was written for those graphics library functions used by the program. The emulation was restricted to only supporting 3D filled polygons because no 2D, 3D line, and text functions were supported in the PC Reality graphics library[1]. We had no ability to implement the PHIGS functions ourselves because we did not have an 80860 development environment available. The flight simulator Flight[2] was successfully ported to the PC; however, the Flight cockpit displays were not used because of the lack of 2D and text functions.

[1]These additional graphics capabilities were promised for a later release of the library.

[2]Recall, Flight is a subset of the Dog software with the only difference between the two programs being the network connections provided by the Dog program.

The virtual world interface work was started by interfacing joysticks to the program for aircraft throttle and stick control as well as some program options selection. A user menu was not implemented because of the missing text support. The Polhemus sensor was interfaced to let the simulator view angle be controlled by the user's head motion. The final interface to the HMD was not accomplished because the PC Reality board could *not* provide interlaced display output.

At this point, program testing (described in section 5.3) showed such poor performance that we determined that networking to the SGI 4D would provide little additional value to the project. The network code had been compiled and executed, but did not work correctly. The Flight software used a lower level UDP/IP (User Datagram Protocol/Internet Protocol) programming interface than the transport layer currently supported by ESIX. ESIX explicitly supports programming at the transport layer of the Open Systems Interconnection model, but not at the network level used by Flight. These network level functions are present in ESIX System V but are undocumented. Some work would have been required to track the incompatibility between the Silicon Graphics' use of UDP/IP and the proper ESIX approach. We felt the time would be better spent developing the new 3D cockpit for the flight simulator.

5.9 Capability Assessment

The 20 MHz Intel 80386 computer, enhanced with a graphics subsystem running two Intel 80860 RISC processors only provided a frame update rate of four (4) frames per second, well below the 15 frame per second goal established for the implementation.

Some timing tests were recorded for three different test program implementations[3]. All programs executed C code compiled with the GNU ANSI C compiler. This code

[3]The first two programs were *not* subsets of Flight. These programs used PC Reality graphics library calls directly.

had a static view point and showed a runway with airport buildings. An aircraft (a C150 Piper Cub consisting of 105 polygons) was shown taking off from the runway. The objects in this program consisted of a sum of 260 flat-shaded polygons, all of which were visible. An aircraft running down the runway was the tightest possible looping construct that could show some form of animation. The program simply calculated a new aircraft position and redrew all the objects. This program provided a frame update rate of 12 frames per second. The second program replaced the C150 object with a detailed description of an F14 consisting of 1034 polygons. The total polygons in the scene summed to 1217. This time, the update rate was only 6 frames per second. The third configuration was a scaled down version of the SGI Flight program showing only an out-the-window display of the runway, runway buildings, mountains, and hills. This program had a total of 485 polygons and was the configuration that provided the 4 frame per second update rate.

From these results, one can deduce that the system is both polygon limited[4] and program limited[5]. The graphics subsystem with the two 80860 processors is polygon limited because of the time required to traverse the PHIGS CSS and render the object descriptions. This was demonstrated by the slow down in the second test program with 1217 polygons from the first with only 260. The third program – the flight simulator with 485 polygons in the CSS, only achieved a frame rate of 4 per second. This slow-down, when contrasted against the 6 frames/sec of program two with 1217 polygons, can only be caused by the 80386/387 system being program limited. This hypothesis is more fully demonstrated later when the final performance measurements for the finished program are presented (see Table 5.3).

5.3.1 Final Simulator Test Results. After optimization, the finished program had better performance than the initial tests demonstrated. Before summarizing the

[4]The maximum number of polygons that can be rendered by the PC Reality board before significant slow-down occurs.

[5]The maximum number of instructions executed on the 80386/80387 processors before significant slow-down occurs

results of the timing measurements, polygon counts for each object (Table 5.1) and sum totals of the polygons contained in the test scenes (Table 5.2) are presented. This will allow the reader to cross reference the timing performance with the approximate number of polygons visible on the screen at any one time.

Table 5.1. Object Polygon Count

Object	No. Polygons	Object	No. Polygons
F14D	1034	F16	186
C150	105	Buildings	34
Hills	302	Mountains	28
Runway	121	Cockpit	41
Infinite World	12		

Timing measurements were recorded in several program configurations with a variety of scenes to provide different polygon counts and PC program conditions. Separate measurements were taken for the HUD view[6] and cockpit view, with and without the Polhemus sensor input. The wingman and tower views provided additional polygons to be displayed in the scene without increasing the processing requirements of the PC processors. These additional polygons allow the observation of timing slow-downs contributed solely by the PC Reality board. Using the recorded measurements (Table 5.3) and associated polygon counts, the hypothesis that the PC Reality board is polygon limited can be verified. The plot in Figure 5.1 shows that as more polygons are added to the CSS, the frame rate continues to decrease.

The timing measurements also confirm that the system is program limited. When the cockpit is added, a significant slow-down occurs. This slow-down cannot be attributed to the additional 41 polygons contained in the cockpit description because we have a slower frame rate, with *less* polygons, than what was measured

[6]The HUD view is an out-the-window only view with no instrumentation. The 4-bit overlay plane of the PC Reality board was not accessible with Version 3.2 of the PC Reality graphics library.

Table 5.2. Total Polygons in Different Views

View	Hills		No Hills	
out-the-window	497		195	
cockpit and out-the-window	538		236	
tower and	Cockpit		No Cockpit	
wingman	Hills	No Hills	Hills	No Hills
C150	602	300	643	341
F14D	1531	1229	1572	1270
F16	683	381	724	422

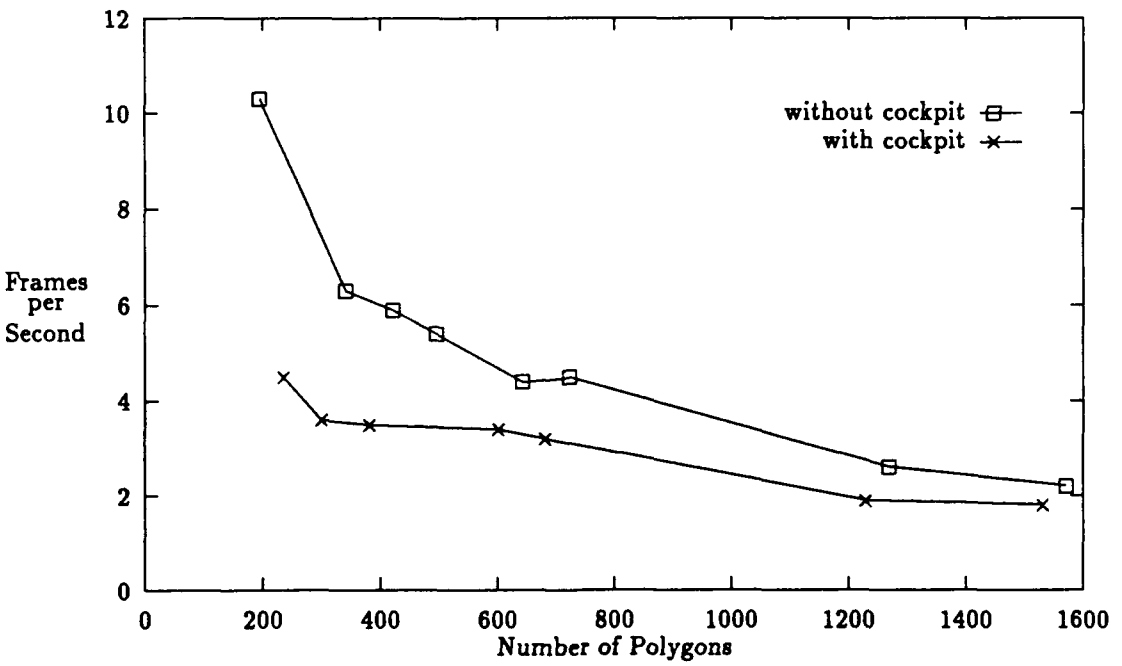


Figure 5.1. Frame Rate VS Polygon Count

Table 5.3. Performance Test Results

A/C	Cockpit	Polhemus	Wingman	Tower	Hills	No Hills	Runway
					(frames/sec)		
C150	on	on	–	–	2.2	2.4	2.3
	on	–	–	–	4.1	4.5	4.3
	–	on	–	–	4.1	4.3	4.3
	–	–	–	–	5.4	10.3	7.4
	on	on	on	–	1.9	1.9	1.9
	on	on	–	on	2.1	2.2	2.2
	on	–	on	–	3.4	3.6	3.4
	on	–	–	on	3.6	3.8	3.6
	–	on	on	–	3.2	3.5	3.5
	–	on	–	on	4.2	4.3	4.3
	–	–	on	–	4.4	6.3	5.9
	–	–	–	on	6.5	7.0	6.5
F14D	on	on	–	–	2.2	2.3	2.2
	on	–	–	–	4.1	4.5	4.3
	–	on	–	–	4.2	4.4	4.4
	–	–	–	–	4.9	9.9	9.4
	on	on	on	–	1.3	1.3	1.3
	on	on	–	on	1.4	1.5	1.4
	on	–	on	–	1.8	1.9	1.8
	on	–	–	on	2.0	2.0	2.0
	–	on	on	–	1.9	2.0	1.5
	–	on	–	on	2.1	2.2	2.1
	–	–	on	–	2.2	2.6	2.2
	–	–	–	on	2.5	2.7	2.5
F16	on	on	–	–	2.2	2.4	2.4
	on	–	–	–	4.0	4.5	4.3
	–	on	–	–	4.1	4.4	4.4
	–	–	–	–	4.9	9.8	8.4
	on	on	on	–	1.7	1.9	1.9
	on	on	–	on	2.0	2.1	2.1
	on	–	on	–	3.2	3.5	3.3
	on	–	–	on	3.5	3.7	3.5
	–	on	on	–	3.1	3.3	3.4
	–	on	–	on	3.9	4.0	3.9
	–	–	on	–	4.5	5.8	5.5
	–	–	–	on	5.7	6.5	6.2

for the out-the-window wingman view with the F16. The cockpit contains 7 objects that are updated every frame. The transformations for the objects are built using the local matrix stack on the 80386/387 and then loaded into the matrix array storage contained on the PC Reality board. Since we have not observed significant slow-down caused by transferring data over the PC bus in other tests, the slow-down must be occurring from the matrix multiplies executing on the PC processors. Plots are shown when the program is in HUD mode, without a cockpit or cockpit instrumentation, and with the cockpit. The plot with the cockpit shows that a frame-rate bias between corresponding polygon counts exist. This slow-down bias has occurred because of the program limitation.

The PC system was also observed to be input/output (I/O) bound (limited). The addition of the Polhemus sensor[7] significantly slowed the frame rate. As can be observed from the timing measurements in Table 5.3, the Polhemus sensor interface approximately halved the frame rate. The measurements show that a performance penalty, approximately equal to that contributed by the cockpit, is present. When both the cockpit and Polhemus are used, an additional linear slow-down occurs. Processing of the Polhemus information only added three additional transformation operations using the local stack. The resulting composite transformation matrix is not passed separately to the PC Reality board but is composed with the existing view matrix that is passed when the Polhemus is not used. From this, one can conclude that the addition of the Polhemus causes a slow-down for reasons other than numerical processing demands on the 80386/387. To further isolate the effects of the Polhemus to the PC processors, note that the wingman and tower views did *not* add additional slow-down when the Polhemus was active. This result occurred because of the parallel structure of processing units (80860 and 80386/387). Use of the Polhemus is a very I/O intensive process with messages constantly being passed from the sensor to the PC. The PC reads the input buffer once each frame. Often

[7]Recall that the Polhemus sensor is used to track head motion for the virtual world interface.

times, the buffer is not at the beginning of a message and the PC must read a second time to receive a valid message. Since the processing demands of the Polhemus sensor information does not add a significant numerical processing demand, the only logical conclusion is that the system is I/O bound. The Silicon Graphics system used in the cooperative thesis effort did not demonstrate similar slow-down when using the Polhemus interface.

5.4 Hardware Integration

Hardware integration on the PC platform provides an additional argument against the suitability of using PCs as a part-task simulator's CIG. PCs do not come with a graphics engine suitable to the task; therefore, the platform must be built using a third party board. This third party board will receive independent support from the PC. This complicates the system integration and subsequent support structure for these platforms. Appendix D contains a summary of the system integration difficulties encountered during this thesis.

5.5 Assessment

We cannot make the blanket claim that the PC is unsuitable for hosting flight simulators. Companies like MicroSoft (Flight Simulator IV) and Spectrum Holobyte (AT-Falcon) have certainly shown that some relatively good flight simulators can be hosted on standard PC configurations. These programs are written in assembly language and make extensive use of bit-block manipulations to achieve such good visual performance. However, as we said from the beginning, software portability across platforms was one focus of our research. Assembly language implementations are platform specific and require major reprogramming efforts to port the application to a different instruction set architecture or video interface.

The cost of high-end PC platforms enhanced with powerful graphics engines are close to mini-computer prices of much greater performance capability. The PC

platform used for this research cost approximately $31,000 without discounts. This breaks out as follows:

$ 7,000	20 MHz Compaq Deskpro/20 with 80387, 120MB hard drive, 4MB memory, ethernet card, VGA card and color monitor, 19 inch NTSC monitor.
$ 20,000	PC Reality board with 2 processors and 4MB of memory.
$ 3,300	PC Reality DOS *or* UNIX driver.
$ 800	ESIX System V operating system with software development extensions, 2 user-license.

The Silicon Graphics IRIS 4D/85 GT costs approximately $50,000 without discounts.

I have shown that using a high-end PC as a CIG does not provide satisfactory performance for the flight simulator. Flight simulators can be hosted on PCs, but they require a great deal of custom programming in assembly language and are not portable across platforms. The more generalized approach of programming the flight simulator in a high order language using standard 3D graphics methods has only been successfully demonstrated on the higher-priced mini-computer systems.

We have been able to clearly demonstrate the performance difference between a PC with a state-of-the-art graphics engine and a Silicon Graphics 4D implementation of the same flight simulator in side by side tests. The Silicon Graphics machine provides a frame update rate of 15 – 20 frames per second for Z-buffered Gouraud-shaded polygons. The PC was only able to achieve 10.3 frames per second for Z-buffered *Flat*-shaded polygons in a scaled down version of the simulator (no cockpit display). This gives us a price performance ratio of $2,500 per frame/sec for the Silicon Graphics 4D versus $3,019 per frame/sec for the PC platform. That makes the PC approach more costly per frame/sec than the seemingly more expensive mini-computer approach.

One could argue that the higher performance Super Reality board with 4 Intel i860's on-board would about double the performance of our system. This would

have achieved an approximate frame rate meeting our goal for the out-the-window only view, but it would have also raised the price another $12,000. Even with this price increase, one would argue that the price performance ratios are now close enough to accept the PC approach[8]. In time, Real World Graphics will improve their support for the additional rendering attributes, and they have promised to improve the throughput of the graphics pipeline.

The PC Reality board has the architecture to support fast graphical transformations. The board handles all the object transformations, the view clip, and display pipeline. Strangely, Real World Graphics chose not to use the i860s to perform all the transformation operations. The creation of the homogeneous modeling transforms (i.e. rotate, scale, and translate) and the viewing transforms (i.e. evaluating the view and map transformation matrices) are done on the host computer. Although these operations are small when contrasted with the number of multiplies that occur when actually transforming an object description, they can cause unexpected processing bottlenecks. An example of one bottleneck is evident from the slow down noted when the 3D cockpit code is executed (see Table 5.3). Unfortunately, we do not have the tools to measure where most of the transformations are taking place (where the processing bottleneck occurs). Even if we did, we had no means of controlling where these transformations were occurring since we only had an 80386 development system, not an i860 development system.

An interesting theory called *the wheel of reincarnation* was presented by T.H. Myer and Ivan Sutherland(34). Their theory was formulated as a result of their development of a graphics display channel. What they found was that as more and more functionality was added to a graphics coprocessor, the more it looked like a general purpose processor. The more it looked liked a general purpose processor, the more the user tended to use it as one. The added loading slowed the graphics

[8]Note that the Silicon Graphics system is displaying its graphical objects as Gouraud-shaded versus our Flat-shaded objects.

coprocessor until another requirement for a fast coprocessor was established, thus *the wheel of reincarnation.* One conclusion from their work was that

> the display processor should be closely coupled with the parent computer, that it should take its data from the main computer's core, and that the user should have complete, bit-by-bit control over that data.(34)

The SGI graphics pipeline uses this principle and allows the original description of the graphical data object to be modified in main memory and then sent into the graphics pipeline during every frame displayed on the screen. This allows the programmer to dynamically control the content of the data object description to directly change its shape or attributes at any time. The object is then re-rendered from main memory whenever it must be displayed. The SGI graphics library provides a rich set of bit-level control mechanisms to control the object descriptions passing through the graphics channel.

Contrasting that approach was the PC Reality system with its separate memory and constrained interface. The PHIGS graphics standard allows user defined functions to be implemented at any level to enhance the functionality of individual products. What the standard achieves is a required minimum functionality. Real World Graphics' omission of bit-level control mechanisms for object descriptions and separate storage memory was a hindrance to easily creating dynamic visual effects. One example was turning on and off the backfacing polygons option to allow a visually appealing wingman view. The PC simulator was not able to achieve this type of dynamic change because of the difficulty of having the 80386 edit static data structures in an external memory. Although the PC Reality board contained 4 MB of memory, the memory partitioning was not defined. The user memory available to hold object descriptions in the CSS was not sufficient to load all of the standard object descriptions normally loaded with Flight. The PC Reality memory can hold the structures for the infinite world, runway, buildings, hills, mountains, and one air-

craft F14D aircraft. When another large aircraft description is sent to the CSS (like the F15), the PC Reality database overflows. All the object descriptions, local stack storage, and executable program successfully function in the 5 MB of PC memory. Myer and Sutherland suggest that the design of a dedicated graphics channel should be based on using a common memory with the host processor. This ability to use a common memory would have provided better programmer control over the CSS.

5.6 Conclusions

After working with the Flight software that was designed for a graphics co-processor using a common memory with the host, and with the PC Reality board that contained its own separate memory, I conclude, from an application programmer's point of view, that partitioning the system memory into internal and external storage is not optimal. The common shared memory recommended by Myer and Sutherland(34) is significantly easier to work with. With the separate memory approach, the objects had to be read into PC memory first, then preprocessed into the correct CSS structure format. Once preprocessed, the structure or structure networks were then sent to the PC Reality to be added to the CSS. This method caused unnecessary program and memory inefficiency and took control of the data structures away from the programmer. From an object oriented approach to programming, this encapsulation of the data structure is desirable, but only when a *complete* set of constructors and selectors are available.

Although I cannot say that PC's are inappropriate as CIGs for flight simulators, a valid claim is that a PC using the PC Reality Board, configured with two i860s, and using version 3.2 of the Reality Graphics Library is unsuitable. A fully functional flight simulator modeled in the classic 3D graphics approach requires more processing power than the 80386/387 host and PC Reality were able to provide. I am also not willing to conclude that some PC configuration can never meet the performance requirements, but a *lot* more effort must be expended to attain the same level of

performance obtained from workstations.

Mid-range mini-computers providing 16 – 20 MIPS continue to come down in price. The high-end PCs continue to improve in performance and go up in price. At this point in time, using a mini-computer is the only reasonable approach for hosting a virtual world flight simulator system that could be used as a part-task training system.

5.7 Recommendations

My recommendations for follow-on research focus on identifying the lowest-cost CIG that will provide a sufficient update rate to support creating a virtual world part-task trainer.

I recommend discontinuing the use of the PC platform in future research. The configuration used for this thesis is well suited as a fast renderer but has been shown to be unsuitable for real-time applications. Real World Graphics' claims their theoretical limits can support real-time applications, but the demonstrated limits have identified that the board and it's associated graphics library are not mature enough to meet the required performance needs. The existing board must have an 8 to 9-fold performance improvement to meet a 30 frames per second update rate for the scaled down version of Flight-PC.

Other low-cost platforms exist that may provide a better performance capability than the Real World Graphics PC Reality board and PC platform. Further work should be directed to other low-end mini-computer systems that border on desktop class systems. The new Sun SPARC 1+ stations have a networked flight simulator that could provide a good foundation for a similar project to this. The NeXT color computer systems are lower cost than the Sun SPARC, with comparable through-put specified, but do not provide a flight simulator. Writing a virtual flight simulator for the NeXT station may be one project that could be performed; creating a virtual world interface for the flight simulator on the SPARC stations could be another.

Additional development work to network flight simulators together is recommended. We had hoped to network the PC to the Silicon Graphics to demonstrate one-one-one and two-ship flight formations. The goal was a little too ambitious for the time allotted. Porting Flight to the PC proved to be far more difficult than initially estimated because of the change in graphical reference models. Extending the interactive flight simulator work into a networked flight training system is the logical next step. We had considered adopting the Army's Simulation Network (SIMNET[9]) protocol as the networking protocol. Continued work investigating and demonstrating the capability of the SIMNET protocol to support a high speed flight simulator could provide a foundation for a possible inter-service simulator. Many skeptics have stated that SIMNET protocols can't support high speed flight simulations. There appears to be no solid evidence that this is true. Demonstration of the capability would clearly set the precedent for continued work.

5.8 Summary

A PC platform was integrated that provided a reasonable high-end processing platform for hosting a flight simulator program. The PC was a 20 MHz 80386 AT with 80387 floating point support. This basic system was enhanced with a Real World Graphics PC Reality board that was driven by two Intel 80860 RISC processors.

We ported the flight simulator Flight hosted on a Silicon Graphics IRIS 4D/85 GT to the PC. This flight simulator is modeled using classic 3D graphics methods and written in the high level programming language C.

An emulation was written for function calls from the SGI graphics library to the Real World Graphics PHIGS-like graphics library. Hardware enhancements available on the SGI machines, optimizing their graphics pipeline and input/output

[9]SIMNET is a networked Army tank simulator with many nodes simultaneously participating

device management, were unavailable on the PC forcing work-around solutions. The solutions introduced some overhead processing into the emulation library.

A virtual world interface was built for the simulator. The interface consisted of joysticks allowing throttle and stick control of the aircraft and a head position tracking system to control the simulator view. A Polhemus sensor was used to allow tracking the position and orientation of a user wearing the Air Force Institute of Technology's second generation head-mounted display system.

Rehosting the flight simulator Dog to the PC platform used for this research is *not* a viable approach for a low-cost part-task trainer. The best update rate we were able to achieve was 10 frames per second from an out-the-window view without the runway, buildings, or other terrain visible. We found that the program was both polygon limited (how fast the PCR board could redraw the objects) and program limited (how fast the 80386/387 could process the flight dynamics). The end conclusion was that the visual results obtained from the PC implementation were unsatisfactory. Mid-cost workstations providing 16-20 MIPS processor performance provide a superior visual display and better price performance ratio than the PC approach.

One contribution from this work was demonstrating that a transformation from the Silicon Graphics graphical reference model to the PHIGS model can be done. The transformation was not simple because of the hardware enhancements of the SGI machine, but all needed functions could be emulated.

Appendix A. Dog's *Object Description File Format*

This appendix describes the object description file format used by the Silicon Graphic's programs Dog and Flight. These object file definitions are used to store the geometric descriptions of the aircraft, runway, buildings, hills, and mountains seen in the program.

A.1 General Format Description

The object description files consist of three parts: 1) a branch list, 2) a transformation list, and 3) an object geometry description or geometry list. Each part will be described in detail in the detailed format description section.

Comments can be included in a description file. Comments are identified by preceding a line with a # sign. Comments can be placed anywhere in a file *except* after a transformation value, geometry point, or vertex index line. Placing a comment after one of these lines will cause the read routine to abort the program with a syntax error message.

One of the best ways to gain a deeper understanding of the object file format is to take a simple object description and walk through the code contained in the file read.c in the flight/lib/libgobj directory.

A.2 Detailed Format Description

This section explains in detail the required format for each part of the file format.

A.2.1 Part I: Branch Nodes. The branch nodes provide four pieces of information: 1) the state of the object (object identification), 2) the mode of the object (drawing attributes e.g. backface drawing on or off), 3) the number of transformations to apply to the object descriptions in this branch followed by a list of indexes

into the transformation list, and 4) the number of branches and leaves extending from this node followed by a list of indexes into the branch or geometry lists.

The first line of this part contains an integer value identifying how many branches are contained in the branch list. This value is used when parsing the object description into memory. All branch node entries following this line consist of three lines.

The first line contains the state and mode of the object to be drawn separated by a comma. Branch zero is the first branch and is always at the root of the tree to be executed.

State Bits. The state bits identify the section or sections of the object to be drawn and whether backfacing polygons are drawn. If the user wants to cull backfacing polygons, bit 31 (or high order bit) of a long integer must be on. The lower 16 bits of the state are usually masked to all 1's since branch zero describes all parts of the aircraft to be drawn. Therefore, the Branch 0 entry for backfacing polygons being culled would read 0x8000ffff.

Subsequent branch state bits identify the part or parts of the geometry being represented by the branch. The state bits are defined in the file object.h. Figure A.1 identifies the basic components of the plane state bits for describing aircraft parts.

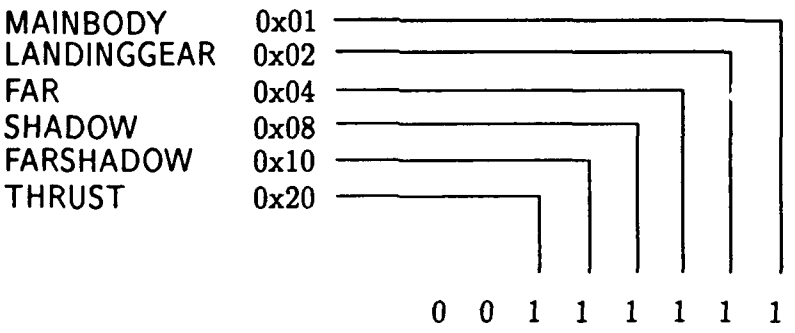


Figure A.1. Plane State Bits

To represent a complete aircraft body with landing gears the state bit would read 0x3, or a combination of the MAINBODY and LANDINGGEAR.

Mode Bits. Mode bits are used to turn on or off drawing features of the individual branches. Four modes exist: 1) backface culling, 2) transparency, 3) Z-masking and 4) color masking. Each of these modes can be turned on or off for a particular object. For example, if a transparent cockpit is being drawn, the user may wish to turn off the culling of backfacing options and turn on transparency. These modes are effective for the immediate mode of drawing into the SGI graphics pipeline, but ineffective for the PHIGS approach. For example, in the SGI machine when the backfacing polygons culled option is turned off (show back facing polygons) and transparency is turned on, backfacing polygons of objects falling behind the cockpit will be seen. In the PHIGS approach they will not be seen because of the static feature of object descriptions in the Central Structure Store. To dynamically turn on the backfacing polygons in objects described in a PHIGS CSS, each object falling behind the transparent object would have to be identified, the structure opened, edited, closed and redrawn. Another approach would be to delete the structure from the CSS, then reload the object with the new attributes. The Z and color masks are specific to the SGI graphics hardware pipeline capabilities.

A.2.2 Part II: Transformation List. The first line of this part is an integer value that lists the total number of entries in the transformation list.

The subsequent lines list the transformation type, followed by the required transformation values for the rotate, scale, or translate that will take place. Comments cannot be placed after the second or subsequent lines of this part.

For more detailed information about how the transformations are effected, see the file draw.c in the libgobj directory.

Table A.1. Transformation Identification

ROTX	0
ROTY	1
ROTZ	2
TRANSLATE	3
SCALE	4

A.2.3 Part III: Geometry List. Each geometry part, except the CDV_GEOM, consists of four items: 1) a section description identification, 2) a material or color value, 3) the number of vertex points followed by the list of vertices, and 4) the number of polygons followed by the list of indices into the vertex list. After these four sections are explained, the CDV_GEOM format is identified.

Line 1: Section Description Identification. The objects are drawn into the SGI graphics pipeline using a variety of different methods depending on whether the object is a filled polygonal object, flat or smooth shaded, or colored; or a line or point object. The possible alternatives are shown in Table A.2.

Table A.2. Geometry Section Identification

SSECTION	5	Smooth Section
FSECTION	6	Flat Shaded Section
PSECTION	7	Planar Section
CSECTION	12	Colored Section
CLS_GEOM	18	Colored Lighted Section
CDV_GEOM	22	Colored Point

Line 2: Material or Color Value. Line 2 means different things for the different geometry types. The meanings are broken out as follows:

For SSECTION, FSECTION, and PSECTION line two identifies the index into a local materials table. The materials are defined in the file light.c in the libgobj

directory. The materials define the color and lighting attributes of the geometric component. The allowable material values are listed in Table A.3.

Table A.3. Material Identification

MAT_SWAMP	1	MAT_GRAY11	15
MAT_PLANE	2	MAT_GRAY12	16
MAT_DIRT	3	MAT_THRUSTER	17
MAT_GRAY0	4	MAT_GLASS	18
MAT_GRAY1	5	MAT_PROP	19
MAT_GRAY2	6	MAT_BORANGE	20
MAT_GRAY3	7	MAT_BLIME	21
MAT_GRAY4	8	MAT_BTAN	22
MAT_GRAY5	9	MAT_BGRAY	23
MAT_GRAY6	10	MAT_PURPLE	24
MAT_GRAY7	11	MAT_LPURPLE	25
MAT_GRAY8	12	MAT_F14BLACK	50
MAT_GRAY9	13	MAT_F14YELLOW	51
MAT_GRAY10	13	MAT_WHITE	52

For CSECTION and CLS_GEOM, line 2 represents a packed hexadecimal representation of the color to use when drawing the polygon. The packed color line is a long integer that represents the alpha channel value (transparency), and green, blue and red values. Hexadecimal representations for the value of each color desired are represented in two hexadecimal digits for each color. The format of the packed long integer looks like *aaggbbrr* with each color value representing a number between 0 and 255. A bright red object would be described with the number 0x000000ff and a bright blue object with 0x00ff0000.

Line 3: Vertex List. For all geometry types except PSECTION and CDV_GEOM, line 3 is an integer value identifying the number of points to follow. For PSECTION, line three is the single normal identified for the entire planar structure description with, line 4 identifying the number of points to follow.

Line 4+ lists the appropriate vertex information based on the type of the geometry. Comments cannot follow these vertex data lines without causing a syntax error when the program is reading the information. The SSECTION contains both vertex point and normal coordinates for each point specified. This is necessary for proper shading. FSECTION through CLS_GEOM types only include the vertex point coordinates.

Line 5: Polygon List. Line 5 is an integer value identifying the number of polygons in the polygon list.

Line 6+ lists different information based upon the geometry type. For SSECTION, PSECTION, CSECTION and CLS_GEOM, these lines represent indices into the vertex list forming a polygon. Each line represents one polygon. For the FSECTION, only one polygon is needed for each polygon, so the normal coordinates precede the vertex reference list.

Table A.4 summarizes the data types included in the vertex data lines and the polygon list.

CDV_GEOM. Line one identifies the geometry type. Line two lists the number of points to follow. Line 3+ lists a packed color value for the point and the vertex coordinates for the point. Line 4 identifies the number of point structures, and line 5 gives the index into the points list. If a single point source is drawn, the polygon structure value is one and the index into the list is zero.

Table A.4. Vertex Information Lines

SSECTION	Points and Normals	Vertex Reference List
FSECTION	Points	Normals and Vertex Reference List
PSECTION	Points	Vertex Reference List
CSECTION	Points	Vertex Reference List
CLS_GEOM	Points	Vertex Reference List
CDV_GEOM	Packed Color and Points	Points Reference List

Appendix B. *Joystick Design and Functions Library*

The ESIX System V operating system did not provide a device driver for a PC game controller card. After installing the UNIX operating system on the PC, Capt Ed Williams wrote a device driver and an Application Programmer's Interface (API) library for interfacing with the driver.

This section describes the joystick device driver and API design. Source code for the joystick device driver is included at the end of this appendix. The game controller board drives both joysticks through the same port.

We had to get additional documentation from ESIX's System V Release C *Release Notes* (we are using release D) to get the information needed to write the device driver. The information required was contained in Appendix C of those release notes.

B.1 Joystick Subsystem Description

We purchased two C&H Products Flight Sticks and an 8-bit game controller card for the ISA bus. Each Flight Stick has a separate trim wheel that is intended to be used as a throttle control in a single joystick system. This wheel is disabled when two joysticks are connected to the port via a **Y** connector.

The joystick consists of two RC (Resistor/Capacitor) circuits; one at each axis. These RC circuits utilize a potentiometer to establish the resistance value of the circuit based on the position of the joystick handle. As the handle is moved around, the potentiometers are adjusted which changes the decay time of the capacitive charge in the RC circuit. This voltage decay is what the game controller detects and provides as bits in a control register. Unfortunately, the potentiometers have slightly different ratings and are generally nonlinear which make the delay times nonlinear.

B.2 Joystick Device Driver

The system was configured to use two joysticks; therefore, the trim wheel was not included in the design. If only one joystick is used, the trim wheel is reported as the Y axis of the second joystick.

The driver had to accommodate very rapid access to allow reading the joystick values without significantly slowing down the overall system operation. The flight simulator's main loop had to execute every 50 ms to obtain a graphics update rate of 30 frames per second. The actual time spent reading the joystick port had to be a small percentage of that 50 ms.

The information provided by the game controller card is TTL (transistor to transistor logic) compatible signals. When the RC voltage discharges to a level less than 0.7 volts, the bit value in the control register drops to zero. If the voltage level raises to above 2.7 volts, the bit value changes to one. The control bit assignments are shown in Figure B.1.

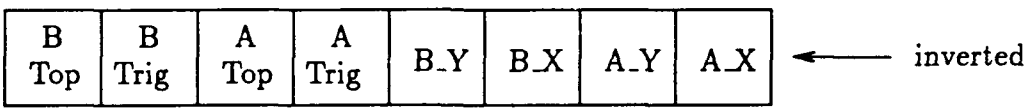


Figure B.1. Joystick Control Register Bits

Capt Williams developed two different approaches to reading the port. The first tried to stabilize the fluctuating instantaneous time values of the RC voltage decay. Unfortunately, the read took approximately 30 to 40 ms per axis. After being told of the 50 ms loop requirement, Capt Williams exploited the true power of the C programming language and manipulated everything at the bit level. Through a sequence of bitwise masks, XORs, and ORs, he was able to read the values of the four potentiometers (two in each stick) and the button pushes in a total of 2 ms or less.

The basic design of the joystick driver follows:

```
output any value to the port setting all timer flags to one
loop
     read control registers
     if any bits are zero record current loop count to determine time
          and record which bits were zero
     increment loop count
     continue until all four joystick flag bits are zero
end loop
Make another read to get button pushes
invert button flags
shift right by 4-bits
Put buttons and axis values into buffer
return
```

Note that the button values will be a value between $0 - 2^4$ depending on which combination of joystick buttons are depressed when the buttons are read. Also, note that the position outputs to the joystick control registers are in an inverted state from a normal binary read or all ones when the value is zero. We are looking for a voltage drop to indicate a joystick axis has been read; therefore the joystick axes bits merely serve as flags to show that the RC circuits have decayed. The actual time of the RC voltage time decay is registered by the loop counter, with each count being approximately 10 μs.

The device driver is a standard UNIX System V device driver and requires the same support files as any other device driver. These support files are gc.node, gc.dev, and Master. The ESIX release notes have further information on installing device drivers.

B.3 Joystick Routines

The low-level joystick routines include:

- A joystick initialization routine
- Joystick read routines
- Button read routines

- A user calibration routine

An initialization routine opens the joystick device and puts the file descriptor in the file local variable gc_fd. The initialization routine is automatically called if the programmer uses the calibrate_stick routine.

A read routine provides a scaled joystick position for both joysticks and a composite decimal value that must be interpreted to determine which combination of buttons was depressed. A series of macros have been defined for the most common joystick button reads. One joystick read returns the five values (stick 0 x and y, stick 1 x and y, buttons value) in a structure with five members.

A user calibration routine is available which should be used during the initial setup of an application program. Starting with stick zero (the left joystick) this routine has the user strike the enter key when the stick is in the center position registering the raw value (RC voltage decay time). The user then moves the stick to the upper left and presses the top joystick button registering the raw values for this corner. The joystick is then positioned in the lower right corner and the top button is pressed again. This calibration operation is repeated for stick one (the right joystick). These raw values are then translated and scaled depending on the user defined joystick range variables.

B.4 Joystick Functions Library

Capt Williams designed several desirable features into the joystick API. These included:

- Center oriented calibration
- User-defined coordinate system
- A center dead zone
- A jitter parameter

The raw values read from the joysticks did not provide a linear set of values centered around the (0,0) position. Instead, the center position was offset somewhat as shown in Figure B.2.

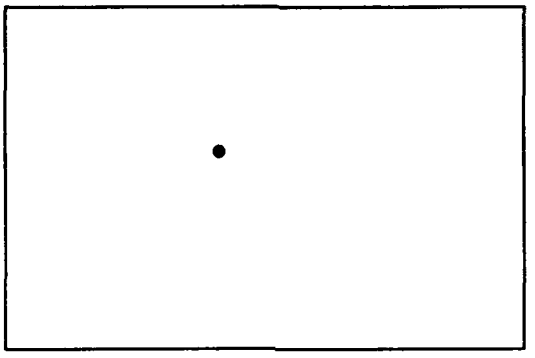


Figure B.2. Raw Joystick Range

Since all the functions of a flight simulator are center oriented, Capt Williams developed a means to make the joystick range a user-definable range (such as from -100 to +100 or 0 to 1000) and centered about the origin like that shown in Figure B.3.

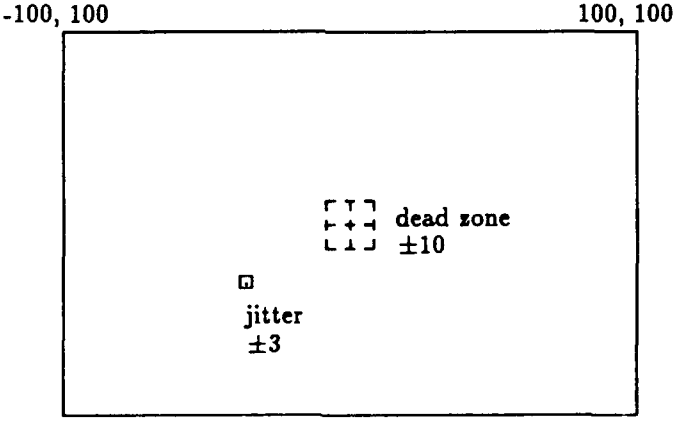


Figure B.3. User Defined Joystick Range

The programmer definable center dead zone variable exists to help prevent readings from a hand resting on the joystick in the center position and accidentally

pushing the stick slightly in any direction. A nominal dead zone of no more than three is recommended for normal use.

The programmer definable jitter parameter also exists to help smooth out instantaneous variations caused by slight differences in reading the RC voltage decay. This jitter parameter always surrounds the current location of the joystick position. A nominal jitter value of no more than one or two is recommended for normal use.

B.5 Game Controller Driver Source Code

Header File.

```
/*
 * gc.h
 */

#define True  (1 == 1)
#define False (0 == 1)

/*
 * some definitions to include kernel info from system header files.
 */
#define KERNEL 1
#define defined_io 1
#define NOSTREAMS 1

/*
 * structure definition for the return data
 */

typedef struct
{
  unsigned short a_x, a_y,
                 b_x, b_y;
           char buttons;
} STICK_DATA;
```

Source File.

```
/*****************************************************************************
*
* Copyright (c) 1990 Air Force Institute of Technology
*        This is FREE software.  NO WARRANTY.
*
*****************************************************************************/
```

```
/*****************************************************************************
*  FILE    : gc.c
*  PROJECT : Virtual World Flight Simulator
*  VERSION : 1.0
*  DESCRIPTION: Device driver for the joystick game controller under
* ESIX System V operating system.
*
*  FUNCTIONS:
* gc_read(dev)
* dummy()
*
*  DATE WRITTEN: 30 August 1990 by Ed Williams
*  LAST MODIFIED:
*  HISTORY:
*     $Log$
*
*****************************************************************************/

#include <sys/types.h>
#include <sys/param.h>
#include <sys/dir.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/user.h>
#include <sys/sysmacros.h>
#include <sys/conf.h>
#include <sys/file.h>
#include <sys/buf.h>
#include <sys/tty.h>

#include "gc.h"
#define BUTTON_MASK 0xf0;

static unsigned tout = 500;
static char flag;

dummy()
{
  flag = 0;
}

/*****************************************************************************
*  METHOD NAME:  gc_read(dev)
*  DESCRIPTION:  Reads the control register bits and returns raw timing
* counts of the time it took for the joystick RC voltage to decay.  Also
* reads the button bits, shifts them right 4-bits and returns the
*  register value.
*  PARAMETERS:  STICK_DATA data; <The resulting data is output to the device>
*  RETURNS:  void
*  GLOBAL VARIABLES USED:  .
```

```
*  DATE WRITTEN:   30 Aug 90 by Ed Williams
*  DATE MODIFIED:  .
*  NOTES:  .
*  HISTORY:
*     .
*
*****************************************************************************/

gc_read(dev)
dev_t dev;
{
  STICK_DATA data;
  register int s;
  register unsigned bits, c;
  unsigned stack[8];
  register int i;
  register unsigned j;

  i = 0;
  bits = 0x0f;
  outb(0x201, 0);
  s = spl7();
  for(j = 0; j < tout && bits; j++)
  {
    c = inb(0x201) & 0x0f;
    if(c ^ bits)
    {
      stack[i++] = c ^ bits;
      stack[i++] = j;
      bits &= ~(c ^ bits);
    }
  }
  splx(s);
  data.buttons = (~inb(0x201) >> 4) & 0x0f;
  i -= 2;
  while(i >= 0)
  {
    switch(stack[i])
    {
      case 1:  data.a_x = stack[i+1]; break;
      case 2:  data.a_y = stack[i+1]; break;
      case 3:  data.a_x = stack[i+1];
               data.a_y = stack[i+1]; break;
      case 4:  data.b_x = stack[i+1]; break;
      case 5:  data.a_x = stack[i+1];
               data.b_x = stack[i+1]; break;
      case 6:  data.a_y = stack[i+1];
               data.b_x = stack[i+1]; break;
      case 7:  data.a_x = stack[i+1];
               data.a_y = stack[i+1];
               data.b_x = stack[i+1]; break;
```

```
        case 8:  data.b_y = stack[i+1]; break;
        case 9:  data.a_x = stack[i+1];
                 data.b_y = stack[i+1]; break;
        case 10: data.a_y = stack[i+1];
                 data.b_y = stack[i+1]; break;
        case 11: data.a_x = stack[i+1];
                 data.a_y = stack[i+1];
                 data.b_y = stack[i+1]; break;
        case 12: data.b_x = stack[i+1];
                 data.b_y = stack[i+1]; break;
        case 13: data.a_x = stack[i+1];
                 data.b_x = stack[i+1];
                 data.b_y = stack[i+1]; break;
        case 14: data.a_y = stack[i+1];
                 data.b_x = stack[i+1];
                 data.b_y = stack[i+1]; break;
        case 15: data.a_x = stack[i+1];
                 data.a_y = stack[i+1];
                 data.b_x = stack[i+1];
                 data.b_y = stack[i+1]; break;
      }
      i -= 2;
    }
    copyout(&data, u.u_base, sizeof(STICK_DATA));
    u.u_base += sizeof(STICK_DATA);
    u.u_count -= sizeof(STICK_DATA);
}
```

Appendix C. *Detailed Design Notes*

This appendix presents specific methods used to emulate the more important functions required to implement the flight simulator software. This information is presented to document some of the design decisions and emulation rationale used during this research. Hopefully this information will be useful to any follow-on development using a similar platform.

C.1 Graphical Reference Model Parameters

Modeling and viewing parameters are not common between the PHIGS graphical reference model and the SGI graphical reference model. At times, two or more SGI functions must be called before all the parameters are initialized for a single PCR PHIGS function call. Because of these differences, I liberally used global variables for the modeling attributes and viewing parameters used by the PCR implementation of the PHIGS model. Some indexing methods also required distribution between more than one SGI function thus qualifying them for global definition.

C.2 Drawing Flight *Geometry Objects*

A library of user defined routines are included with the Flight software. These routines provide functions to read a geometry file into memory and then to send the geometry description to the SGI graphics pipeline. These functions are included in the library directory libgobj. The method used to read the geometry description into memory was acceptable for our implementation. My focus was on how to extract the geometry information from the data structure in memory, and put it into a static structure description used by the PC Reality PHIGS Central Structure Store (CSS). Each object entered into the CSS needed to be capable of accepting transformations to allow independent motion (such as a plane landing on an airfield or landing gears being raised). To facilitate this requirement, a pointer to a global and local transfor-

mation matrix which had been initialized to the identity matrix was inserted as the first two elements of every structure. The PCR board has a table of transformation matrices (0-49) stored in local memory. By reserving indexes into this table of matrices for different types of objects (like aircraft, aircraft parts – wings and wheels, and other objects) each object could be independently moved at a later time by modifying the transformation matrix in the table. During a previous design approach, I had inserted the matrices into the structure rather than just the pointers. In order to move the objects, the structure had to be edited to replace the existing matrix with a new one. This was a time consuming process and much less efficient than the current implementation. The matrix index for the local transformation matrices of aircraft are defined to be 0 through 9 and can be accessed through the #define established for each aircraft. For example, the C150 is defined as 0 (zero) and is also the index into the table of matrices that is the local transformation matrix for the C150. The global transformation matrix index is accessed through the aircraft #define number + 10. A #define value has been set up for each aircraft that has been put in objects.h by preceding the aircraft name with a G for global (i.e. #define GC150 (C150 + 10)). For static objects like the buildings, runway, mountains, etc., the local matrix is OBJECTS_IX and the global is GOBJECTS_IX. When transformations are being done that do not affect any object, a NULL_IX should be used. The transformation methods (rotate (rot), scale, and translate) are tied to the table of transformation matrices. If the position of the C150 needs to be updated, the global variable trans_ix must be set equal to C150, then call the transformations you want to apply to the aircraft. As a safe programming practice, the trans_ix should be set to the NULL_IX when the object transformations are completed. For transformations that you do not want reflected in the table of matrices, set or leave the global trans_ix to the NULL_IX. NULL_IX is defined to be 50 which is off the end of the table of matrices. An if statement captures any trans_ix > 49 and does not post it into the table.

The SGI call to drawobj() works differently than how I was able to emulate the function. A call to readobject() stores the geometric description of the aircraft with any local transformations into SGI system memory. Each time through the main loop of the flight simulator a call to drawobj() is made for each object displayed in the simulation. This call starts a traversal of the data structure redrawing the aircraft into the graphics pipeline. Any modifications to the object or list of transformations is traversed and executed as if it were the first time the object were being drawn. This approach cannot be used when using the PHIGS CSS. On the PC, the object structure is read into local PC memory using the readobj() call. This description is then parsed and sent to a similar PHIGS hierarchical description in memory on the PC Reality board. For each aircraft part, an independent structure is created. At the beginning of each of the component structures a local transformation matrix is inserted with any l.. :al transformations that position the parts (such as the wheels). Note that for aircraft parts, the local transformation matrices are inserted into the structure itself rather than just an index into the table of matrices. This was done because there are not enough matrices in the table to accommodate all unique parts. The aircraft is then put together in the hierarchy of the CSS by creating a structure network. A global structure element is created with the indices into the table of matrices and then an RExecute() call for each part of the aircraft. The only method of changing the description is to delete the entire description and read in a new description (a slow process) or to post changes by editing the static structure store in PCR memory.

As a result of the static representation of the aircraft in PCR memory, the drawobj() calls were moved out of the main loop of the flight simulation. They were drawn as a preprocessing step prior to the main loop. Note that only local modeling changes needed to be edited into the static structure stores (such as for retracting wheels or sweeping wings). Positioning of the aircraft was done similar to the methods used for the SGI graphics pipeline by transforming the aircraft position

from the origin to its current location and orientation in each pass of the loop. These changes in position and orientation of the entire aircraft were achieved through modifying the transformation table as described above.

To support drawing polygons into the CSS, an emulation of the SGI high-performance drawing mode was required. The high-performance drawing mode provides the fastest means of drawing primitive graphical figures on a 4D series machine. The old IRIS method of defining an object using makeobj() is still supported (39:2.35, 16.2) but is less efficient than the new high-performance mode routines. The new mode defines points, lines, and polygons in terms of vertices. A point is a single vertex, a line segment is two vertices identifying the end points, and a polygon is a set of three or more vertices identifying the corners. To draw a graphical object in this mode, a series of vertex subroutines must be used – surrounded by a pair of begin and end subroutines marking the beginning and end of the object definition (39:2.2).

To emulate the SGI n3f() (normals) and v3f() (vertices) functions, different array addressing schemes had to be implemented – one for the normals and one for the vertices. n3f() updates all subsequent normals for vertices sent to the graphics pipeline so it can be used to update all the normals of a flat shaded polygon or set each subsequent normal for a GOURAUD shaded polygon. v3f() sends the polygon vertices directly to the graphics pipeline. Each of these functions have only one parameter which is the address of the normal or vector array to send to the graphics pipeline. This information has to be stored into a polygon array to be able to use the PCR RConvexFill() function. Since the array indexing is accomplished external to the two functions, a local static variable was used to keep the array indexing progress, one variable for each function. Several #define statements were developed to accommodate correct incrementing and resetting of the static variables.

To write polygon descriptions into the CSS, I chose to use an incremental RConvexFill() approach rather than the more eloquent RVertexRefList() approach because there is a 100 vertex limit on any one object definition (37:B.42). The **flight**

program has several objects that have more than 100 vertices defined. The RConvexFill() approach parses vertex arrays into polygon definitions and then sends them to the PC Reality memory as individual polygons. The RVertexRefList() approach would have allowed direct use of the object reference definitions in the Dog object definitions files.

The SGI lighting model requires three components to be able to perform a lighting calculation: 1) a surface material, 2) a light source, and 3) a lighting model. These are set by using the lmdef() and lmbind() functions (39:9.36 - 9.47). The lmdef() function can define all three of the components in the SGI lighting calculation. The definitions of the various components were stored in local arrays for use with PC Reality function calls that partially implement the SGI functions. SGI methods allow independent specification of ambient, diffuse, and specular color reflections for materials whereas the PC Reality does not. The lmbind() function emulation presented a challenge because the function can be used in the SGI graphics pipeline at any time. Many of the comparable functions in the PC Reality graphics pipeline (such as setting object surface properties) could only be executed when a static CSS structure was built. Careful use of the functions had to be observed to prevent violating the PC Reality graphics pipeline rules.

C.2.1 Viewing Reference Model Emulation Flight only uses two graphics library function calls that effect the viewing parameters: viewport() and perspective(). A third function emulation for lookat() was also developed to attempt a replacement for a custom Flight function named my_lookat(). This last function is used for the tower view.

viewport() defines the left, right, bottom, and top edges of the viewport, defined in screen coordinates, as arguments to the function. We are restricted to NTSC mode, thus the maximum screen coordinates are 640 X 484. To emulate the viewport() call in PHIGS, the parameters were normalized by dividing by the maximum

screen values. These values were then loaded into the first four positions of the six position viewport array required by the PCR library function REvalViewMap-Matrix3(). The last two positions of the array define the minimum and maximum viewport limits in z and are loaded with the minimum (-1.0) and maximum (0.0) values in normalized projection coordinates (NPC). The other parameters required by the function REvalViewMapMatrix3() are defined by the SGI function perspective(). The PRP is defined by default as (0,0,0) – the center of the VRC system – for a viewer oriented view. The PRP can be changed by directly assigning new values to the global floating point array prp[]. REvalViewMapMatrix3() is called in both the viewport() and perspective() function calls because of the distribution of the the PHIGS required variables between the two functions. If viewport() is called before perspective() default values established in global variables are used.

perspective() specifies a viewing pyramid into the world coordinate system. The parameters define the field-of-view angle, in tenths of degrees, in the y direction; an aspect ratio which determines the field-of-view in the x direction; and the near and far clipping planes specified in *positive* distances from the viewer. The PHIGS viewing model allows more degrees of freedom than the SGI viewing model. To align the two different models, some of the PHIGS degrees of freedom had to be fixed. One of these was the view plane distance (VPD). This distance, in conjunction with the PCR model's *window* coordinates, and front and back plane distances, establish the shape of the perspective view volume. The correct VPD for Flight was determined by testing images on the screen and adjusting the VPD until the same perspective was achieved on the PC display as on the SGI display. The correct VPD for Flight was -500.0. Note the distance is specified in negative distance rather than positive distance. Both the SGI and the PCR PHIGS view reference models use right-handed coordinate systems and both define the viewing volume down the negative z axis[1]. The near and far parameters of the perspective() function call are

[1]The viewing volume surrounds the $-Z$ axis

specified in positive distances. The function emulation is accomplished by using the parameters to establish the PCR PHIGS window parameters. These parameters define the four corners of the view window located at the VPD from the viewer (see figure 4.3). The sides of the viewing volume are defined by rays emanating from the PRP and passing through the four corners of the view window. Figure C.1 illustrates

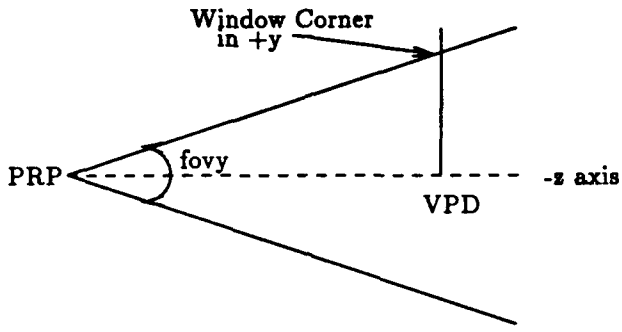


Figure C.1. View Parameters

that the dimensions of the window is a ratio of half the view window width in y to the distance from the center of projection or PRP to the view plane. By fixing the distance to the view plane, we can then control the width of the view plane with the perspective() field-of-view and aspect parameters. The front and back plane distances are the negative of the near and far plane specifications.

C.3 Device Queueing

The SGI graphics library supports three classes of input devices: valuators, buttons, and other devices. Valuators provide integer input from devices such as the horizontal and vertical position of the mouse. Buttons return a boolean value identifying whether they are pressed or not. Other devices include the keyboard key presses or combination of key presses.

SGI provides two means for getting values from input devices: polling or queueing. Polling gets a value from the device whereas queueing uses an event queue to

save changes in device values and other input events so the program can read them later. Although both polling and queueing are used in Flight, only queueing affected our emulation.

Devices that are queued act as asynchronous devices independent from the user process. Whenever a device changes state, an entry is made in the event queue. The user specifies which device will be queued in the event queue. Each entry into the event queue includes the device number and a device value. Although the SGI input queue can contain up to 101 events at the same time, the emulation limits the queue to 64 events.

The SGI event queue allows the user to specify which devices to queue and then allows only those devices to enter changes in state to the event queue. The emulation of their event queue isn't quite as robust. The event queue emulation restricts input devices to the keyboard. However, the programmer has the flexibility for manually queueing events into the event queue. The event queue emulation provides a qdevice() function only for library compatibility. The function serves to initialize the emulation event queue. Other queueing functions provided in the emulation include qenter(), qreset(), qtest(), and qread(). All functions but qread() work identical to the SGI functions. The SGI qread() will block until a device queues – the emulation does not block. The rest of the SGI event queue functions were not applicable to supporting Flight and were not implemented.

The event queue emulation is a circular queue reading inputs from the keyboard. Each time through the main loop in the Flight program, a call to qread() is made which checks to see if there have been any keys depressed at the keyboard, if there have, then it enters the device number and key value. Key values are not equivalent to the same key values used by the SGI keyboard. Reference the include file keys.h for specific key values assigned for the AT-386 keyboard definition.

Because the event queue emulation does not block, a simple change to the Flight function wait_for_input() was required. The function now waits for a key depress and

throws away the value of the key.

C.4 Communications Software

Dog uses the User Datagram Protocol and Internet Protocol (UDP/IP) for its network interface. This code uses much older function calls to implement the protocols at the network layer instead of the transport layer of the Open Systems Interconnection (OSI) model[2]. Although the older functions still exist in the ESIX library, they are undocumented.

The UDP service provides a transport-level datagram service. ESIX reports that this protocol is basically an unreliable service, with delivery and duplication protection not guaranteed(10:2/4). Like the more common Transport Control Protocol (TCP), UDP is expected to work with the Internet Protocol. UDP assembles a data unit and hands it to IP for transmission. There is no error provision and only a one-way handshake is used. Invalid data units are checked with a passed checksum with invalid units simply being discarded.

My attempt at using the existing network code failed. With the tools available, I was able to collect enough information to surmise that the socket was not being established correctly, in non-blocking mode, at the network layer. To implement a network connection, the code would have to be rewritten from scratch using the newer transport level calls documented in the ESIX documentation[3].

[2]For a description of the OSI model see the Network/STREAMS Programming Guide(10:1/1).
[3]The Silicon Graphics documentation also supports only the newer transport level calls.

Appendix D. *Thesis System Integration*

Working within the fiscal constraints of this thesis, the system integration plan consisted of obtaining parts and components from a number of different sources. The system configuration was defined to be an 80386 PC platform with:

- 100 Megabyte or larger Hard Disk
- 5 Megabytes of memory
- Graphics engine providing NTSC output
- Standard VGA driver for software development
- VGA monitor
- NTSC color monitor
- Ethernet card
- Multi-IO card with serial and parallel interfaces
- Joystick card
- 2 joysticks

Trying to maintain compatibility with the cooperative thesis effort being developed on the Silicon Graphics IRIS 4D 85/GT, the operating system was defined to be UNIX system V.

The basic PC platform with VGA monitor and a 19 inch NTSC monitor was borrowed from the Human Resources Laboratory, Operations Training Division (OTE) at Williams AFB. The machine they provided was a Compaq Deskpro 386/20 with 1 Megabyte of RAM, a multi-IO card with one serial and one parallel port, and a 65 Megabyte Hard Disk.

A few components were borrowed from within the institute. These were a 120 Megabyte hard disk, an ethernet card, and a mouse.

The graphics engine was purchased from Simulation Technologies Inc. acting as the United States sales agent for the United Kingdom's Real World Graphics Ltd.. Real World Graphics Ltd. manufactures the board and developed the graphics library interface to the underlying hardware.

The two joysticks and joystick card were purchased from a local vendor.

Finally, I purchased the ESIX System V operating system which is quite a robust package with a low price tag (less than $1000).

Now all that remained was to integrate the parts and get a working system. Integration would simply consist of plugging in all the required PC boards, loading the ESIX system V operating system with their install program and start my development work.

D.1 Integration Problems

Many problems were encountered, these problems are identified only to highlight the dangers of system integration. The lesson I hope to convey is that system integration is an important, but difficult part of an application development; a part that might be better left to a vendor.

The first difficulty started with the borrowed PC platform from Williams AFB. A number of problems were encountered with this platform; most could have been avoided if I had more knowledge about the non-standard configuration of Compaq machines.

The basic computer platform selected for the development provided a number of problems. The system was a borrowed resource and as I learned through experience, a non-standard platform.

The first problem encountered was difficulty getting the system shipped from OTE. The point of contact (POC) at OTE wanted a formal letter from the School of Engineering at AFIT identifying the transfer of equipment. This letter had to be

processed through headquarters (HQ) Air Force Systems Command (AFSC) which then required inter-command coordination between AFSC and Air University (AU). The result was that the system was requested in February 89, but did not leave Williams AFB until July 89.

This delay required other needed parts to be ordered ad hoc from my guess of what was required. Compaq Corp. technical services requires system users to deal directly with authorized local dealers for technical questions or problems concerning Compaq computers. The sales staff at the two authorized local dealers in Dayton did not have comprehensive knowledge about Compaq systems. Each time I called a dealer, the sales representative tried handling my technical question. Each time I received advice, follow-up work showed the information to be erroneous. I did not discover that these local dealers had more knowledgable service representatives available until much later; unfortunately, the service representatives did not possess all the information needed either.

The system was shipped from OTE after involving some new people, these individuals distinguished themselves by getting the system into shipping channels within a week and shipping it Air Logistics which got the system to me within another week.

Unfortunately, the basic platform was damaged in transit. The high voltage line in the VGA monitor and the 65 Megabyte hard disk were damaged. AFIT/SC distinguished themselves by their outstanding support to get the system fixed. They added the system hardware to their repair support contract. The VGA monitor was picked up the day after it arrived and sent off for repair. The hard disk failures were intermittent and we finally succumbed to the problem after about a week of fighting with it. We sent the whole system out for repair through the AFIT support subcontractor. The monitor came back repaired about 4 weeks after it was sent and the computer came back about 6 weeks after it was sent. Unfortunately, the hard disk was still working intermittently after repair and was not reliable enough to use.

I was not able to initially use the 120 Megabyte MFM (Modified Frequency Modulation) Hard Disk that I had borrowed from AFIT/ENG. The Compaq uses IDE (Integrated Device Electronics) disk drives and disk controllers. I had assumed it used an MFM controller. This was one disadvantage of trying to *guess* a system's configuration. I had asked OTE what size hard disk was in the machine but didn't think to specifically ask what type of hard disk and controller was used. I wasn't too concerned about the lack of disk space because ESIX system V provided Remote File System (RFS) support to other system V units. The Silicon Graphics 4D had plenty of disk space and was running system V unix. I didn't find out until late August that Silicon Graphics only provides the Network File System (NFS) protocol and not the RFS protocol — the hard disk size became an issue again.

With the 65 MB IDE drive still malfunctioning, and no way to use RFS support for additional disk space, I had to consider alternate hard disk configurations. The most obvious solution was to just buy a new 100 MB (or larger) IDE hard disk. Unfortunately, these cost in excess of $800. Our funds had already been depleted. The lowest cost alternative was to disable the hard disk control portion of the Compaq hard disk controller (leaving the controller to control the floppy disk) and install an MFM hard disk (only) controller. The 120 MB disk could then be installed solving all the storage problems. This was the option chosen. After buying the card from a local Compaq vendor, we could not get it to work correctly in the system. We eventually carried the system to the vendor and had them install the card. Fortunately, the plan worked and we were able to install UNIX on the system.

The ethernet card borrowed from AFIT/ENG did not work out either. After studying the ESIX system V release notes, I understood the severity of my oversight in reading the sales literature. The sales literature had listed ESIX support for numerous vendors. That list turned out to be an all inclusive list. The release notes identified that only components from those vendors were supported and only certain models of those vendors equipment were supported. AFIT/ENG had three

different manufacturers ethernet cards that they could provide me, none of which were supported by ESIX. Some more homework identified a Western Digital ethernet card as the cheapest compatible card available. Good fortune arrived, as I was calling to order the card, I saw an unused card on a table near the phone. I asked the owner if I could borrow it and he was gracious.

The joysticks and driver card presented two problems. First, ESIX provided no drivers for joystick cards. Fortunately, Capt Ed Williams (a UNIX expert) volunteered to write any required drivers needed. Second, August arrived and the joysticks still had not arrived. The company we ordered the components from had sold out to another national company and the order got lost somewhere. After three weeks of follow-up we finally connected with the right person and an order was shipped. We ordered two joysticks (throttle and stick) and one controller and received one joystick and two controllers. The company gladly corrected their mistake, but it cost another three weeks time to fix the error.

The PC Reality Board integration also caused problems. The board was shipped with the default address set to 0240 hexadecimal (H) while the documentation stated it was set to 0300H. We detected the incorrect addressing and set jumpers to install the board. The board was first installed on a Zenith 248 system under DOS while the Compaq was out for repair. After successfully integrating the board on the Z-248 platform, I developed a viewit program to test the capabilities of the Reality Graphics Library. After a working Compaq system was returned, the board was installed to run under UNIX. Real World Graphics had provided a generic UNIX driver to build into the kernel. Capt Ed Williams built the kernel but had to patch the base address of the board in the Reality Graphics Environment (RGE). Real World Graphics had established the base address as 200H instead of 0300H claiming their was a UNIX device driver at 0300H. There is no device driver at 0300H, so using the Norton Utilities program – Norton Disk Doctor, the addresses in the RGE were patched to 0300H. The default values in the Reality Library API are

set to 0300H, had the 0200H addressing scheme been used,the base address of the board would have needed to be reset by the software every time a program was executed. After successfully integrating the board and the driver, I attempted to port the **viewit** program developed under DOS using an earlier version of their library. The program didn't work. RGE version 3.2 required the use of an undocumented command open_reality_comms() before making the call to RGEOpen(). This undocumented command was discovered by inspecting the source code in a demonstration program provided with the PC Reality UNIX driver.

September 1, 1990 marked the first day of having a full hardware platform ready for serious software development. The myriad of hardware and integration problems had seriously slipped the development schedule.

I strongly recommend that if a follow-on thesis is organized after the Compaq system is returned to Williams AFB, a fully configured PC platform with the UNIX operating system installed, be purchased from a local vendor. This will facilitate local support if problems develop. The only integration issue remains with the PC Reality board or any other graphics engine not supported by a single vendor.

Appendix E. *Flight-PC Operating Instructions*

Flight is the Silicon Graphics' flight simulator provided for their IRIS 3 and 4 architectures. Keith Seto, technical manager for Silicon Graphics Inc. (SGI), gave special permission to port the flight simulator to a single PC system in support of thesis work being conducted at the Air Force Institute of Technology. The flight simulator was modified to use a virtual interface that consists of a head mounted display for viewing and joysticks for aircraft control. This appendix provides a brief description of the command line options and operating instructions for the PC version of the Virtual Flight Simulator (VFS) derived from Flight.

E.1 Background

The PC uses a state-of-the-art graphics coprocessor to drive the single graphics channel of the VFS. This graphics engine was the PC Reality purchased from a British Firm Real World Graphics Ltd. through their U.S. subsidiary Simulation Technologies Inc.. The PC Reality is based on Intel's new RISC processor architecture - the 80860. The board uses two 80860s, one acts as a preprocessor to the second processor. Accompanying the board was a graphics library based upon the PHIGS PLUS standard. The library was a Beta version and lacked much of the functionality defined by the complete standard. The functions that were available supported building a Central Structure Store[1] (CSS) and 3D concave filled polygons. The missing functions[2] forced several features of the program to be lost in the translation.

Text and 2D functions were under development by Real World Graphics, but not in time for our use in this prototype development. In addition to the missing functions, several deficiencies were encountered with the Reality Graphics Environment

[1] An object database.

[2] 3D lines, 2D polygons and lines, Text, and overlay plane.

(RGE) which loads the board with the executable portion of the 80860 code[3]. We found that the scan-line Z-buffer algorithm could not handle penetrating polygons[4]. This deficiency explains why the user sees the skull and cross bones flickering on and off when viewing the F14D or F15. The runway description was modified to prevent that effect from occurring for all the lines and taxi-ways. The global modeling transform was supposed to move structures relative to the world coordinate system. Real World Graphics' incorrectly implemented this global transform in Version 3.2 of the library by moving objects relative to the View Reference Coordinate (VRC) system rather than world coordinates. Thus, only the local modeling transform was available to use for motion – this is why I didn't implement moving flaps and landing gears. Component movement could have been accommodated, but for a prototype development, wasn't considered essential. Another severe developmental detriment was that I couldn't get the PC Reality graphics library's RLabel() to function correctly. This prevented accessing the specific locations within a structure or structure element contained in the CSS without knowing the explicit position of the element relative to the start of the structure. Lacking a method to read the description from the CSS[5], this task was impossible.

To use the program with the Polhemus sensor, the program flight.c must be recompiled using the -DPOLHEMUS flag set in the Makefile, then relinked with the rest of the programs. This is easily accomplished by uncommenting the flag in the Makefile, executing the UNIX function touch on flight.c and executing make at the command line prompt. When the Polhemus sensor is used, the sensor must be turned on before executing flight.

E.2 Command Line Options

Flight has several command line options (summarized in Table E.1) that execute

[3]We did not have an 80860 development system to modify, correct, or enhance this code.
[4]Silicon Graphics' object descriptions use *many* penetrating or coplanar polygons.
[5]No selector functions are included in the library.

the program using different features or capabilities. The standard Flight options that are functional for this implementation and two new command switches are described.

Table E.1. Command Line Options

-h	HUD Mode	Full Screen out-the-window view with no instrumentation.
-s	Test Mode	Keeps your gas tank full and weapons load full.
-a	Anti-aliasing Mode	Implements the PC Reality anti-aliasing mode – *Very* slow.
-g	Gouraud Shading	Implements the PC Reality Gouraud shading mode – RGE implementation does NOT correctly implement Gouraud shading.

As described in the Table E.1, the HUD mode does not contain any of the 2D instrumentation contained in Flight executing on the Silicon Graphics. The HUD instruments are drawn in an overlay plane on the SGI screen. The 4-bit overlay plane for the PC Reality was inaccessible with RGE Version 3.2. Text and 2D functions were absent also, so the PC screen rather than the high resolution RGB monitor was used for essential text output .

The anti-alias mode works as advertised[6] but slows the system to approximately 1 frame/sec when using the wingman or tower views (see Section E.3 for details on using these views.).

Gouraud shading is implemented incorrectly in the RGE code. Apparently a global surface attribute is applied to all objects in the CSS thus projecting all objects in their true color at full intensity. The objects look smoother due to wash-out rather than prope. shading.

CAUTION: This program cannot be the first program executed after applying power to the PC because of a bug in the Reality Graphics Environment, see the Notes section for further details.

[6]See the Reality PC User's Guide, Issue No. 2.

E.3 Operating Instructions

Both a text display, standard VGA monitor connected to the PC or workstation monitor connected over a network; and a non-interlaced, 30 Hz horizontal scan, high resolution red, green, blue (RGB) monitor with a synchronization signal is required. The graphical output uses NTSC *resolution*[7], not signals.

Initialization instructions, help, and report cards are all displayed on the text terminal used with the PC. The PC is accessible through network connections and can be flown with any kind of workstation connected, as long as the joysticks are available[8]. Not all keyboards map to the same key codes as the AT-386 keyboard. Some functionality may be lost when using different keyboards[9].

The simulator is flown through the use of the joysticks. The left stick (or stick_0) controls the throttle, rudders, flaps, and spoilers. The right stick (or stick_1) steers the aircraft.

On-line instructions are available through the help screen. The help screen is accessible through the keyboard and joysticks. To bring up help, press the 'h' key on the keyboard or press the fire_button on the left stick and both buttons on the right stick simultaneously. Table E.2 provides a summary of the operating instructions for Flight-PC. Other keyboard commands used in the SGI version may also function correctly for Flight-PC but have not been tested.

Flight characteristics do *not* function the same as in Flight-SGI. Wing stalls and G-limits are not enforced and all the aircraft can pull outside loops. The collision detection feature used by the Silicon Graphics systems is not available in the PC version; thus, you can fly through hangars, hills, and mountains.

The PC cockpit display is shown in Figure E.1. Flaps indicators are blue

[7]NTSC resolution is 640 X 484.

[8]Most of the developmental test work was done with the joysticks positioned to either side of a Sun3 connected to the PC via ethernet.

[9]Changing views to over the wings or behind the aircraft were inaccessible when using the Sun3 keyboard, all other functions worked correctly.

Table E.2. Keyboard and VFS Interface

s/a	Thrust (+/-)	stick_0 move +/- y axis
f/F	Flaps(+/-)	both_buttons_0 && stick_0 move +/- y axis
c/C	Spoilers(+/-)	both_buttons_1 && stick_0 move +/- y axis
	Rudders(R/L)	both_buttons_0 && stick_0 move +/- x axis
t	Track/Lock-on	fire_button_0
w	Fire Sidewinder	top_button_1
q	Fire Rockets	top_button_0
e	Fire Gun	fire_button_1
r	Detonate Missile	
l	Landing Gear	both_top_buttons
v	Autopilot	both_buttons_1 && top_button_0
R	Restart End of Runway	all_buttons
L	Restart Landing Pattern	
U	Restart Random Flying	
d	Tower View	
W	Wing Man View	
h	Help	fire_button_0 && both_buttons_1
ESC	Quit	

squares, with each square representing 10° of flaps applied. The spoilers indicator are dark tan squares with each square representing 20° of spoilers applied. The landing gear indicator is a single green square that represents that the landing gear is in the down position. If your airspeed exceeds approximately 400 knots, the flaps and landing gear will be ripped off and a single red square will be displayed at each location.

The thrust indicator has a red bar that represents the thrust level. Hash marks are provided for 25%, 50%, 75%, and 100% thrust. Thrust indicator bar up is forward thrust and down is reverse thrust. Reverse thrust is only available when on the ground.

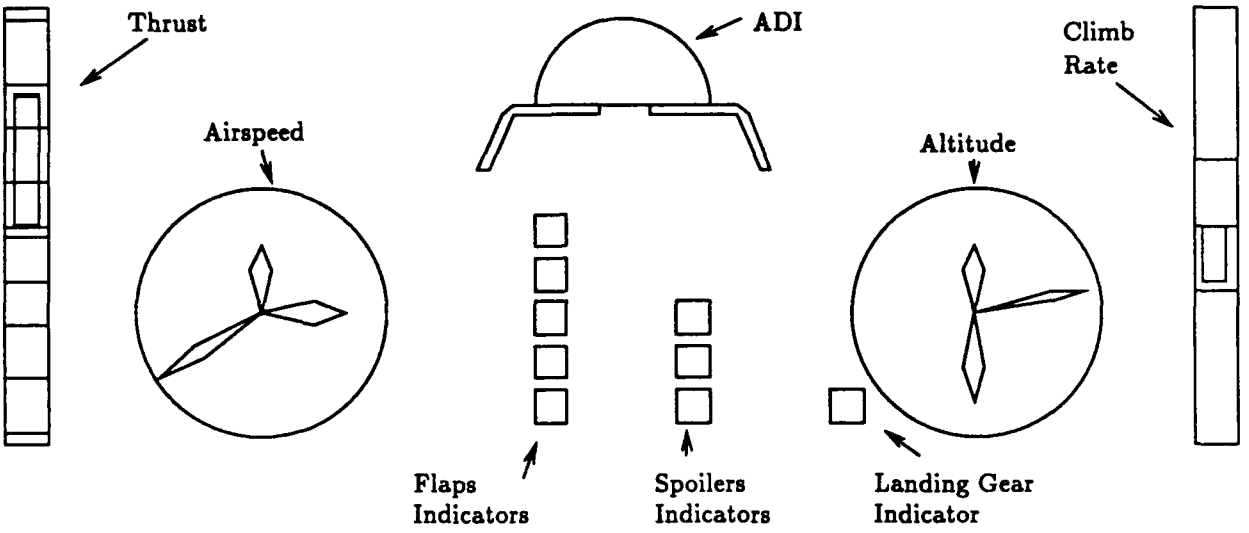


Figure E.1. PC Cockpit Display

The climb rate indicator has a red bar that represents a positive or negative climb rate. The gauge is graduated with less movement near the top and bottom of the gauge representing a much greater rate than movement near the middle. The two hash marks represent + and − 400 fpm climb rate. To land without exploding on impact, you must be descending at a rate less than 400 fpm. Flaps help reduce the descent rate when you are close to −400 fpm.

The airspeed indicator arrows represent knots, hundreds of knots, and thousands of knots of airspeed. The red arrow represents the thousands of knots, the short black arrow – hundreds of knots, and the long black arrow – knots.

The altitude indicator represents hundreds of feet, thousands of feet, and tens of thousand feet. The red arrow represents tens of thousands, the short black arrow – thousands, and the long black arrow – hundreds.

The ADI uses a single rectangle over a blue circle to represent the horizon. At certain transition points you can observe the ADI flipping around. This is an unfortunate effect but also occurs on the Silicon Graphics machine; however, on the

SGI it occurs so rapidly that it is not distracting.

If you are using the wingman view and switch to the tower view, when you try to switch back, the viewpoint is from the wingman's position but the aircraft is not visible. You need to toggle the wingman switch 'W' twice so the aircraft is reposted to the CSS and is visible.

E.4 Notes

The Reality Graphics Environment does not behave correctly if Flight-PC is executed as the first program after initial power is applied to the PC. The flaw is within the RGE and causes processor 0 to *not* initialize. A work-around could have been added to the program to correct the problem, but I determined that the ported program should not be responsible for correcting external errors.

The problem is generated when the emulation for the SGI clear() function is executed. The emulation unposts all structures contained in the CSS at the current time. When the machine is first powered on, the RGE won't accept an RUnPostStruct() or RUnPostAllStruct() function call until at least one structure has been posted to the CSS at some time during the boards powered on history.

A test program named viewit is available that will post a valid structure. Flight-PC will then execute at any time until the system is again powered off[10]. viewit needs an object description, with optional directory path if necessary, to a Silicon Graphics object description (with a '.d' suffix) or an AFIT geometry file (with a '.geom' suffix). The object will be displayed on the screen. If using the PC keyboard, strike CTL-Z to exit. If connected with a networked workstation, you can exit by pressing all four joystick buttons simultaneously. Flight-PC can now be successfully executed. Running any of the demonstration programs provided by Real World Graphics will also post a structure to the CSS and allow correct operation. Their demonstration

[10]since the PC is running UNIX, this should not be a common occurrence.

program flight[11], available in the /usr/local/src/RGE[12] directory, will work. To execute the the Real World Graphics flight program, change to the $RGE/flight directory and execute flight flight. A binary data file is loaded. When the message Downloading flight.rdb . . . done appears, strike the return key to begin execution.

One more note of caution. After pressing the space bar after the help screen, wait a few moments before selecting an aircraft type. After the help screen is displayed, the program continues reading any unread object descriptions. When disk activity stops (which usually occurs after about 5 to 10 seconds) then select the desired aircraft.

[11]Not to be confused with Flight for the Silicon Graphics or Flight-PC ported from the Silicon Graphics.

[12]An environment variable RGE exists to allow the user to change to the directory by executing cd $RGE.

Bibliography

1. Alluisi, Earl A. *Image Generation/Display Conference II, 10-12 June 1981: Closing Comments*. Technical Report, Air Force Human Resources Laboratory, 1981. AFHRL-TP-81-28.
2. Apiki, Steve and others. "The Brains Behind the Graphics," *BYTE Magazine*, *14*(12):178–198 (November 1989).
3. Booch, Grady. *Software Components with Ada*. Menlo Park, CA: The Benjamin/Cummings Publishing, Inc., 1987.
4. Breglia, Denis R. "On-Board Computer Image Generator (CIG) Applications." In *1984 IMAGE III Conference Proceedings*, pages 431–437, 1984. DTIC AD-P004 335.
5. Brooks, Frederick P. *Grasping Reality Through Illusion, Interactive Graphics Serving Science*. Technical Report TR88-007, Department of Computer Science, CB#3175, Sitterson Hall, Chapel Hill, NC 27599-3175: University of North Carolina at Chapel Hill, March 1988.
6. Brooks, Frederick P. and others. "Project GROPE – Haptic Displays for Scientific Visualization," *Computer Graphics*, *24*:177–185 (August 1990). SIGGRAPH '90 Proceedings, August 6-10, Dallas TX.
7. Chung, J. C. and others. "Exploring Virtual Worlds with Head-Mounted Displays," *SPIE*, *1083*:42–52 (1989).
8. Crumley, Lloyd M. *Air-to-Air Gunnery; An Analysis of the Problem, of Present Training, and Recommendations for an Improved F-14 Training Sequence*. Phase Report NADC-75033-40, Naval Air Development Center, April 1975.
9. Ditlea, Steve. "Another World: Inside Artificial Reality," *PC/Computing*, pages 91–102 (November 1989).
10. ESIX Inc. *ESIX System V Network/STREAMS Programming Guide*, 1989. MAN-00475-10.
11. Filer, Robert E. *A 3-D Virtual Environment Display System*. MS thesis, Air Force Institute of Technology, 1989.
12. Fisher, Scott S. and others. "Virtual Environment Display System," *Interactive 3-D Graphics*, pages 77–82 (October 1986).
13. Foley, James D. and Andries Van Dam. *Fundamentals of Interactive Computer Graphics*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1984.

14. Geltmacher, Hal E. *Recent Advances in Computer Image Generation Simulation*. Aviation, Space, and Environmental Medicine 59, Air Force Human Resources Laboratory, Williams AFB, November 1988.

15. Grimes, Jack and others. "The Intel i860 64-Bit Processor: A General-Purpose CPU with 3D Graphics Capabilities," *IEEE Computer Graphics & Applications*, pages 85–94 (July 1989).

16. Grunin, Lori. "Putting Your PC on Tape," *PC Magazine*, *9*(13):197–211 (July 1990).

17. Haber, Ralph Norman. "Flight Simulation," *Scientific American*, *255*:96–103 (July 1986).

18. Hanson, Capt Caroline L. "Fiber Optic Helmet Mounted Display: A Cost Effective Approach to Full Visual Flight Simulation." In *Proceedings of the Interservice/Industry Training Equipment Conference (5th) Held at Washington, DC*, pages 262–268, November 1983. DTIC AD-P003 482.

19. Holzer, Robert. "Engineering Simulation: Reaching New Heights," *Military Forum*, pages 44–50 (November/December 1989).

20. Howard, T.L.J. "An Annotated PHIGS Bibliography," *Computer Graphics Forum*, *8*:262–265 (December 1989).

21. Howard, T.L.J. "PHIGS and PHIGS PLUS Tutorial." In *Proceedings Eurographics '90*, June 1990.

22. Humphrey, Watts S. *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1989.

23. INMOS. *The Transputer Family Product Information*, June 1986. Distributor: Arrow Electronics, Inc., Electronics Distribution Division, 3155 Northwoods Parkway, Norcross, Georgia 30071; Phone: 404/449-8252.

24. INMOS. *Transputer Reference Manual*, January 1987. Distributor: Arrow Electronics, Inc., Electronics Distribution Division, 3155 Northwoods Parkway, Norcross, Georgia 30071; Phone: 404/449-8252.

25. Instruments, Texas, "TMS34020/34082 Graphics Products Preview Bulletin." Preview Bulletin SPVT065, 1988.

26. International Standards Organization. *Information Processing Systems – Computer Graphics; Programmer's Hierarchical Graphics System (PHIGS); Part I: Functional Description First Edition*, March 1989. ISO Standard 9592 PT I.

27. Kanko, Mark A. *Geometric Modeling of Flight Information for Graphical Cockpit Display*. MS thesis, Air Force Institute of Technology, 1987.

28. Kellogg, Robert S. and others. "Simulated A-10 Combat Environment." In *The Image II Conference Proceedings*, pages 35–44, 1981. AFHRL-TR-81-48/DTIC AD A11 0226.

29. "Coup Scored with Reality." The S. Klein Newsletter on Computer Graphics, August 1989. Special SIGGRAPH '89 Double Issue.

30. Kohn, Les and Neal Margulis. "Introducing the Intel i860 64-Bit Microprocessor," *IEEE Micro*, pages 15–30 (August 1989).

31. Lambert, R. E. and others. *Onboard Simulation: A Newly Emerging Technology and the Potential of the Helmet Mounted Display as an Embedded Training Device*. AIAA Flight Simulation Technology Conference, McDonnel Aircraft Company, McDonnell Douglas Corporation, and AFWAL/FIGX, Wright-Patterson AFB, July 1985.

32. Lorimor, Gary K. *Real-Time Display of Time Dependent Data Using a Head-Mounted Display*. MS thesis, Air Force Institute of Technology, 1988.

33. Martin, Stephen W. and Richard C. Hutchinson. *Low Cost Design Alternatives for Head Mounted Stereoscopic Displays*. SPIE Three-Dimensional Visualization and Display Technologies 1083, Naval Ocean Systems Center, 1989.

34. Myer, T. H. and I. E. Sutherland. "On the Design of Display Processors," *Communications of the ACM*, *11*:410–414 (June 1968).

35. Polhemus Navigation Sciences Division, McDonnel Douglas Electronics Company, Cochester, Vermont. *3-Space User's Manual*, January 1985.

36. Real World Graphics Ltd, 5 Blue Coat Ave, Hertford, Shire SG 141PB. *REALITY PC User's Guide* (Issue no. 2 Edition), 1990. US Distributor: Simulation Technologies, 1201 Tulip Avenue, Williamstown, NJ 08094.

37. Real World Graphics Ltd, 5 Bluecoats Avenue; Hertford; Herts SG14 1PB; England. *REALITY PC User's Guide* (No. 2 Edition), 1990.

38. Rebo, Robert Keith. *A Helmet-Mounted Virtual Envrionment Display System*. MS thesis, Air Force Institute of Technology, 1988.

39. Silicon Graphics Computer Systems. *IRIS-4D Series Graphics Library Programming Guide*.

40. Silicon Graphics Computer Systems. *IRIS-4D Series Graphics Library Reference Manual*.

41. Sommerville, Ian. *Software Engineering*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1989.

42. Stoner, Barbara. "The i860: A General-Purpose Solution for Radar Signal Processing," *Military & Aerospace Electronics*, *1*(2):33–34 (February 1990).

43. Sutherland, Ivan E. "The Ultimate Display." In *Proceedings of IFIP 65, vol. 2*, pages 506–508, 582–583, 1965.

44. Sutherland, Ivan E. "A head-mounted three dimensional display." In *1968 Fall Joint Computer Conference, AFIPS Conference proceedings*, pages 757–764, 1968.

45. Texas Instruments, Market Communications Manager, P.O. Box 1443, MS 736, Houston, Texas 77251–1443. *TMS34020 User's Guide*, 1990. Customer Response Center, (800) 232-3200.

46. "Texas Instruments TMS34010 Third Party Guide," January 1989. Third Edition.

47. Wardin, Charles L. *Battle Management Visualization System*. MS thesis, Air Force Institute of Technology, 1989.

48. Woodruff, Robert R. and others. *Advanced Simulator for Pilot Training and Helmet-Mounted Visual Display Configuration Comparisons*. Technical Report AFHRL-TR-84-65/DTIC AD-A155 326, Air Force Human Resources Laboratory, Operations Training Division, May 1985.

49. Yan, Johnson K. *Advances in Computer-Generated Imagery for Flight Simulation*. IEEE Computer Graphics & Applications 5, The Singer Company, August 1985.

Vita

Captain David A. Dahn [redacted] He graduated from East High School in Phoenix. Following high school, he enlisted in the Air Force as a Morse Systems Operator and was assigned to the USAF Security Service (now USAF Electronic Security Command). He spent the next eight years with the Electronic Security Command serving four overseas assignments at 6917 Security Squadron, San Vito dei Normanni Air Station in Southern Italy, the 6987th Security Group, Shu Lin Kou Air Station, Taiwan; 6920th Electronic Security Group, Misawa AB, Japan; and finally the 6913th Electronic Security Squadron, a ground mobile tactical intelligence unit, based at the United States Army's Flak Kaserne in Augsburg, Germany, in 1980. He was selected for the Air Force's Airman's Education and Commissioning Program (AECP) in 1982 and awarded an undergraduate degree in Electrical Engineering from Arizona State University in 1985. David was commissioned after completing the Officer's Training School and was assigned as a computer engineer for the Mission Critical Computer Systems Branch, Engineering Division, for the Air Force Systems Command Armament Division (later renamed the Munitions System Division) located at Eglin AFB, FL. He worked there until he entered the Air Force Institute of Technology (AFIT) to work towards a Masters of Science in Computer Engineering.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	*Master's Thesis*

4. TITLE AND SUBTITLE
A Low-Cost Part-Task Flight Training System:
An Application of a Head Mounted Display

5. FUNDING NUMBERS

6. AUTHOR(S)
David A. Dahn, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/90D-01

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Abstract

Computer Image Generators (CIG) driving high performance flight simulators used in training pilots are expensive. This project investigated whether a small class of these simulators that focus on task specific training could be hosted using much cheaper simulator systems by using a virtual world interface. We implemented the virtual world interface for the Silicon Graphics' Flight program on an 80386/80387 PC-AT enhanced with a high performance graphics engine based on two Intel i860 RISC processors.

One goal was to determine whether the PC environment was mature enough to support our approach. The specific question we tried to answer was whether the flight simulator could be programmed on the PC using a classic workstation approach (written in a high order language using a standard three dimensional graphical reference model). The measure of success was whether the simulator could provide a frame update rate of 15 frames per second or better for Z-buffered, flat-shaded polygons.

The results were short of the requirement. Our conclusion is that the price performance ratio in terms of frames per second was better for the higher priced mini-computer approach than the super-charged PC approach.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Virtual World Environments, Head-Mounted Displays, Low-Cost Flight Simulation	118
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL